П. И. Рудаков, К. Г. Финогенов

ЯЗЫК АССЕМБЛЕРА:

УРОКИ ПРОГРАММИРОВАНИЯ



МОСКВА ДИАЛОГ-МИФИ 2001

УДК 32 973.1
ББК 681 3
Р83

Рудаков П. И., Финогенов К. Г.
Р83 Язык ассемблера: уроки программирования. – М.: ДИАЛОГ-МИФИ, 2001. – 640 с.

ISBN 5-86404-160-2

Книга является простым и доступным для широкого круга пользователей пособием по программированию на языке ассемблера для персональных компьютеров IBM PC. Рассматриваются основы разработки программ, аппаратная организация компьютера, использование системных средств DOS и BIOS, программирование сопроцессора и защищенного режима. Отдельный раздел посвящен разработке прикладных драйверов Windows для управления нестандартной аппаратурой.

Для читателей, не являющихся профессионалами-программистами, но имеющих дело с персональными компьютерами, а также студентов вузов, аспирантов и преподавателей.

ББК 32.973.1

Учебно-справочное издание

Рудаков Петр Иванович, Финогенов Кирилл Григорьевич

Язык ассемблера: уроки программирования.

Редактор О. А. Голубев
Корректор В. С. Кустов
Макет Н. В. Дмитриевой

Лицензия ЛР N 071568 от 25.12.97. Подписано в печать 10.07.2001.
Формат 70x100/16. Бум. газетная. Печать офс. Гарнитура Таймс.
Усл. печ. л. 51,6. Уч.-изд. л. 37,96 Тираж 5 000 экз. Заказ 115

Акционерное общество “ДИАЛОГ-МИФИ”
115409, Москва, ул. Москворечье, 31, корп. 2. Т.: 320-43-55, 320-43-77
Http://www.bitex.ru/~dialog
E-mail: dialog@bitex.ru

Отпечатано на Ордена Трудового Красного Знамени
ГУП Чеховский полиграфический комбинат
Министерства Российской Федерации по делам печати,
телевещания и средств массовых коммуннкаций
142300, г. Чехов, Московская обл.,
тел.: (272) 71-3-36, факс: (272) 62-5-36

ISBN 5-86404-160-2

Предисловие

Книга рассчитана на пользователей персональных компьютеров типа IBM PC, которые хотели бы познакомиться с архитектурными особенностями этих машин, системой команд и режимами работы применяемых в них микропроцессоров, организацией ввода-вывода и прерываний, управлением аппаратными средствами компьютера, функциями базовой системы ввода-вывода BIOS и операционной системы MS-DOS. Все эти знания естественным образом приобретаются при изучении языка ассемблера – языка низкого уровня, приближенного к аппаратным средствам компьютера и его "натуральным" возможностям. В книге будут последовательно рассмотрены основные средства языка ассемблера микропроцессоров Intel и его применение при программировании задач разного рода: вычислительных, логических, по управлению аппаратурой и др.

Как известно, программы, написанные на языке ассемблера (если, конечно, они написаны грамотно), отличаются высокой эффективностью, т. е. минимальным объемом и максимальным быстродействием. Это обстоятельство обусловило широкое использование языка ассемблера в тех случаях, когда скорость работы программы или расходуемая ею память имеют решающее значение. Некоторые классы программ (например, программы драйверов устройств, отличающиеся жесткой структурой) требуют для своего составления обязательного использования языка ассемблера. С другой стороны, поскольку современные системы программирования позволяют объединять в одну выполнимую программу фрагменты, написанные на разных языках, широко практикуется составление комбинированных программ, в которых основная часть программы написана на языке высокого уровня, а наиболее критические участки – на языке ассемблера. Может использоваться и обратный метод, когда в программу на языке ассемблера вставляют фрагменты для выполнения относительно сложных логических или математических преобразований, написанные на языке высокого уровня. Такой метод, в частности, применим при разработке драйверов. Процедуры на языке Си, включаемые в текст драйвера, упрощают программирование и отладку драйвера и ускоряют процесс его разработки.

Однако, кроме потребительских качеств, язык ассемблера имеет еще значительную методическую ценность. Отражая архитектурные особенности и режимы работы используемого в компьютере микропроцессора, язык ассемблера предоставляет уникальную возможность изучения машины на "низком уровне", освоения того, что и как умеет делать аппаратура компьютера и что вносит в его работу операционная система. Знакомство с внутренними возможностями компьютера чрезвычайно полезно, в частности, для программиста, работающего на языках Паскаль или Си, так как позволяет увидеть за формализмом языка высокого уровня те реальные процессы, которые будут протекать в системе при выполнении прикладной программы и, следовательно, более осознанно подойти к разработке структуры программы и ее конкретных алгоритмов.

Язык ассемблера, как и любой другой язык программирования, имеет массу встроенных средств, позволяющих в ряде случаев ускорить и облегчить процесс программирования и расширить возможности создаваемых программ. Профессиональная работа на языке ассемблера, естественно, предполагает детальное знакомство со всеми этими средствами. Чем лучше пользователь владеет техникой программирования на языке ассемблера, тем более эффективными будут его программы. Однако

не менее важной является и другая сторона вопроса – освоение особенностей применения языка для реализации аппаратных и программных возможностей компьютера. В этом плане вопросы, нашедшие отражение в настоящей книге, можно условно разбить на две группы. В первую группу входят сведения по основам языка и программирования на нем, в частности:

- способы адресации;
- система команд;
- типичные алгоритмы программ на языке ассемблера;
- выполнение арифметических и логических операций;
- преобразование данных;
- организация программ и подпрограмм;
- макросредства ассемблера.

Другую группу составляют различные аспекты реализации в программах на языке ассемблера аппаратных и системных возможностей компьютера:

- программное управление аппаратурой;
- создание обработчиков аппаратных прерываний;
- использование прерываний BIOS и функций DOS;
- программирование арифметического сопроцессора;
- работа в защищенном режиме.

Книга рассчитана на самостоятельную проработку ее читателем на персональном компьютере. Уже в первой статье книги дается простейшая программа на языке ассемблера, на основе которой рассматриваются наиболее общие архитектурные вопросы. В дальнейших статьях приводимые примеры программ постепенно усложняются, обрастают все новыми деталями и дают возможность вводить в изложение новые понятия языка и архитектуры компьютера. Авторы сознательно отказались от традиционного абстрактного изложения теоретического материала, заменив его рассмотрением большого количества тщательно подобранных (и, по возможности, простых) программных примеров, которые можно и нужно выполнять на компьютере по мере чтения соответствующих статей.

При составлении книги авторы исходили из того, что для читателя представляет интерес не столько сам язык программирования, сколько его использование для решения различных практических задач. В то же время серьезная работа с компьютером невозможна без глубокого понимания всех аспектов его жизнедеятельности: архитектуры процессора и компьютера в целом, систем ввода-вывода и прерываний, назначения и функционирования периферийных устройств компьютера, функциональных возможностей операционных систем и т. д. Все эти вопросы в той или иной степени освещены в соответствующих разделах книги. Завершают книгу разделы, посвященные разработке прикладных драйверов для систем Windows 95/98 и Windows NT/2000, – материал, слишком специфичный для включения его в традиционные учебники и в то же время представляющий значительный интерес для разработчиков автоматизированных систем управления экспериментальными или производственными установками.

Более детальную информацию по затрагиваемым в книге вопросам читатель сможет найти в интерактивных справочниках, входящих в системы Borland C++, Visual Studio, DDK для Windows 95, 98, NT и 2000, а также по различным адресам в Интернете, в частности www.intel.com, www.amd.com, www.amd.ru, www.microsoft.com и www.vesa.org.

Раздел первый
ОСНОВЫ

Статья 1. **Первая программа**

Начнем изучение языка ассемблера с рассмотрения простой программы, которая ничего не вычисляет и не обрабатывает, а всего лишь выводит на экран терминала строку с фразой "Начинаем!" (пример 1.1). Вопросы ввода в компьютер текста программы и подготовки программы к выполнению мы рассмотрим в следующей статье, а пока сосредоточимся на структуре программы.

Пример 1.1. Простейшая программа

```
text    segment             ;(1)Начало сегмента команд
assume  CS:text,DS:data     ;(2)Сегментный регистр CS будет указывать на сегмент
                            ; команд, а сегментный регистр DS - на сегмент данных
begin:  mov   AX,data       ;(3)Адрес сегмента данных сначала загрузим в AX,
        mov   DS,AX         ;(4)а затем перенесем из AX в DS
        mov   AH,09h        ;(5)Функция DOS 9h вывода на экран
        mov   DX,offset mesg;(6)Адрес выводимого сообщения должен быть в DX
        int   21h           ;(7)Вызов DOS
        mov   AH,4Ch        ;(8)Функция 4Ch завершения программы
        mov   AL,0          ;(9)Код 0 успешного завершения
        int   21h           ;(10)Вызов DOS
text    ends                ;(11)Конец сегмента команд
data    segment             ;(12)Начало сегмента данных
mesg    db 'Начинаем!$'     ;(13)Выводимый текст
data    ends                ;(14)Конец сегмента данных
stk     segment stack       ;(15)Начало сегмента стека
        db 256 dup (0)      ;(16)Резервируем 256 байт для стека
stk     ends                ;(17)Конец сегмента стека
        end   begin         ;(18)Конец текста программы с точкой входа
```

Следует заметить, что при вводе исходного текста программы с клавиатуры можно использовать как прописные, так и строчные буквы: транслятор воспринимает, например, строки text segment и TEXT SEGMENT одинаково. Однако с помощью ключа /ML можно заставить транслятор различать прописные и строчные буквы в именах. Тогда строки text segment и TEXT segment уже не будут эквивалентны. Фактически они будут описывать два разных сегмента с именами text и TEXT. Неэквивалентность прописных и строчных букв касается только имен программных объектов (сегментов, процедур); строки

```
        mov   ds,ax
        MOV   DS,AX
        mov   DS,AX
```

во всех случаях воспринимаются одинаково.

В настоящей книге в программах используются преимущественно строчные буквы. Прописными буквами выделены обозначения регистров, а также имена программных и иных файлов.

Наша программа содержит 18 строк-предложений языка ассемблера. Первое предложение с помощью оператора segment открывает сегмент команд программы. Сегменту дается произвольное имя text. В конце предложения после точки с запятой рас-

полагается комментарий. Предложение языка ассемблера может состоять из четырех полей: имени, оператора, операндов и комментария, располагаемых в перечисленном порядке. Не все поля обязательны; так, в предложении 1 есть только имя, оператор и комментарий, а операнды отсутствуют; предложение 3 включает все 4 компонента: имя begin, оператор (команда процессора) mov, операнды этой команды AX и data и, наконец, после точки с запятой комментарий; в предложении 4 (и многих последующих) отсутствует имя.

Любая программа должна обязательно состоять из сегментов – без сегментов программ не бывает. Обычно в программе задаются три сегмента: команд, данных и стека. В сегменте команд располагается собственно программа, т. е. описание (с помощью команд процессора) последовательности требуемых действий. В сегменте данных описываются данные, с которыми должна работать программа; в нашем примере это строка текста. Назначение сегмента стека будет описано ниже.

В предложении 2 мы с помощью оператора assume сообщаем ассемблеру (ассемблером называется программа-транслятор, преобразующая исходный текст программы в коды команд процессора), что сегментный регистр CS будет указывать на сегмент команд text, а сегментный регистр DS – на сегмент данных data. Сегментные регистры (а всего их в процессоре 4) играют очень важную роль. Когда программа загружается в память и становится известно, по каким адресам памяти она располагается, в сегментные регистры заносятся начальные адреса закрепленных за ними сегментов. В дальнейшем любые обращения к ячейкам программы осуществляются путем указания сегмента, в котором находится интересующая нас ячейка, а также номера того байта внутри сегмента, к которому мы хотим обратиться. Этот номер носит название относительного адреса или смещения. Транслятор должен знать заранее, через какие сегментные регистры будут адресоваться ячейки программы, и мы сообщаем ему об этом с помощью оператора assume (assume – предположим). При этом в регистр CS адрес начала сегмента будет загружен автоматически, а регистр DS нам придется инициализировать вручную. Обращение к стеку осуществляется особым образом, и ставить ему в соответствие сегментный регистр (конкретно – сегментный регистр SS) нет необходимости.

Строго говоря, в приведенной программе, где нет прямых обращений к ячейкам сегмента данных, не было необходимости указывать в предложении с оператором assume операнд DS:data (указание соответствия сегмента команд сегментному регистру команд CS обязательно во всех случаях). Учитывая, однако, что практически в любой разумной программе обращения к полям данных имеются, мы с самого начала написали оператор assume в том виде, в каком он используется в реальных программах.

Первые два предложения программы служат для передачи служебной информации программе ассемблера. Ассемблер воспринимает и запоминает эту информацию и пользуется ею в своей дальнейшей работе. Однако в состав выполнимой программы, состоящей из машинных кодов, эти строки не попадут, так как процессору, выполняющему программу, они не нужны. Другими словами, операторы segment и assume не транслируются в машинные коды, а используются лишь самим ассемблером на этапе трансляции программы. Такие нетранслируемые операторы иногда называют псевдооператорами или директивами ассемблера в отличие от истинных операторов – команд языка.

Предложение 3, начинающееся с метки begin, является первой выполнимой строкой программы. Для того чтобы процессор знал, с какого предложения начать выпол-

нять программу после ее загрузки в память, начальная метка программы указывается в качестве операнда самого последнего оператора программы end (см. предложение 18). Можно подумать, что указание точки входа в программу излишне: ведь как будто и так ясно, что программу надо начать выполнять с начала, а закончить, дойдя до конца. Однако в действительности для программ, написанных на языке ассемблера, это совсем не так! Исходный текст программы может начинаться с описания подпрограмм или полей данных. В этом случае предложение программы, с которого нужно начать ее выполнение, может располагаться где-то в середине текста программы. И завершается выполнение программы совсем не обязательно в ее последних строках, а там, где стоят предложения вызова служебной программы операционной системы, предназначенной именно для завершения текущей программы и передачи управления системе (см. предложения 8...10). Однако начиная от точки входа программа выполняется строка за строкой точно в том порядке, в каком эти строки написаны программистом.

В предложениях 3 и 4 выполняется инициализация сегментного регистра DS. Сначала значение имени text (т. е. адрес сегмента text) загружается командой mov (от move – переместить) в регистр общего назначения процессора AX, а затем из регистра AX переносится в регистр DS. Такая двухступенчатая операция нужна потому, что процессор в силу некоторых особенностей своей архитектуры не может выполнить команду непосредственной загрузки адреса в сегментный регистр. Приходится пользоваться регистром AX в качестве "перевалочного пункта". Кстати, обратите внимание на то, что операнды в командах языка ассемблера записываются в несколько неестественном для европейца порядке – действие команды осуществляется справа налево.

Предложения 5, 6 и 7 реализуют существо программы – вывод на экран строки текста. Делается это не непосредственно, а путем обращения к служебным программам операционной системы MS-DOS, которую мы для краткости будет в дальнейшем называть просто DOS. Дело в том, что в составе команд процессора и, соответственно, операторов языка ассемблера нет команд вывода данных на экран (как и команд ввода с клавиатуры, записи в файл на диске и т. д.). Вывод даже одного символа на экран в действительности представляет собой довольно сложную операцию, для выполнения которой требуется длинная последовательность команд процессора. Конечно, эту последовательность команд можно было бы включить в нашу программу, однако гораздо проще обратиться за помощью к операционной системе. В состав DOS входит большое количество программ, осуществляющих стандартные и часто требуемые функции – вывод на экран и ввод с клавиатуры, запись в файл и чтение из файла, чтение или установка текущего времени, выделение или освобождение памяти и многие другие.

Для того чтобы обратиться к DOS, надо загрузить в регистр общего назначения AH номер требуемой функции, в другие регистры – исходные данные для выполнения этой функции, после чего выполнить команду int 21h, (int – от interrupt – прерывание), которая передаст управление DOS. Вывод на экран строки текста можно осуществить с помощью различных функций DOS; мы воспользовались функцией 09h, которая требует, чтобы в регистре DX содержался адрес выводимой строки. В предложении 6 адрес строки mesg загружается в регистр DX, а в предложении 7 осуществляется вызов DOS.

В предложениях 5 и 7 указанные в тексте программы числа сопровождаются знаком h. Таким образом в языке ассемблера обозначаются шестнадцатеричные (далее – 16-ричные) числа, в отличие от десятичных, которые никакого завершающего знака не требуют. В вычислительной технике чрезвычайно широко используется 16-ричная

система счисления, так как она позволяет в более наглядной форме описывать содержимое регистров или ячеек памяти, хотя значительно затрудняет арифметические операции с этим содержимым. Опишем вкратце основы 16-ричной системы счисления.

Одна 16-ричная цифра может принимать 16 разных значений, первые 10 из которых обозначаются обычными десятичными цифрами, а последние 6 – первыми буквами латинского алфавита от A до F (рис. 1.1).

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Шестнадцатеричные цифры
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Их десятичные эквиваленты

Рис. 1.1. Шестнадцатеричные цифры и их десятичные эквиваленты

Удобство 16-ричной системы счисления определяется тем, что одну 16-ричную цифру можно уподобить 4-разрядному регистру, который также может принимать 16 разных состояний. Поэтому содержимое байта (8 двоичных разрядов) целиком описывается двумя 16-ричными цифрами, содержимое слова (2 байта, или 16 двоичных разрядов) – четырьмя, а содержимое двойного слова (32 двоичных разряда) – восьмью.

Далее, 16-ричное число очень просто перевести в двоичное. Для этого достаточно каждую цифру 16-ричного числа представить в виде четырех двоичных цифр (рис. 1.2).

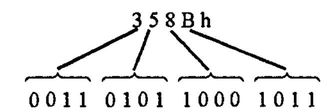


Рис. 1.2. Преобразование 16-ричного числа в двоичное

Обратное преобразование – из двоичной формы в 16-ричную тоже не представляет труда. Надо разбить исходное двоичное число на группы по 4 бита и каждую такую группу представить в виде 16-ричной цифры.

Легко сообразить, что максимальное число, которое можно записать в 1 байт, составляет FFh=255; в 2-байтовое 16-битовое слово можно поместить число не большее FFFFh=65535. Поскольку минимальное число составляет 0, всего с помощью 2 байт можно описать 2^{16} = 65 536 различных чисел. Величина 2^{10} = 1024 обозначается в вычислительной технике буквой K (от кило); таким образом, максимальное значение 16-битового числа составляет 64 K–1.

Вернемся к рассмотрению текста программы. После того, как DOS выполнит затребованные действия, в данном случае выведет на экран текст "Начинаем!", выполнение программы продолжится. Вообще-то, нам, вроде, ничего больше делать не нужно. Однако на самом деле это не так. После окончания работы программы DOS должна выполнить некоторые служебные действия. Надо освободить занимаемую нашей программой память, чтобы туда можно было загрузить следующую программу. Надо вызвать компонент операционной системы, который выведет на экран запрос DOS (как правило, в виде символа >, предваряемого именем текущего каталога) и будет ждать следующей команды оператора. Все эти действия выполняет функция DOS с номером 4Ch. Эта функция предполагает, что в регистре AL находится код завершения нашей программы, который она передаст DOS. При желании код завершения только что закончившейся программы можно "выловить" в DOS и проанализировать, но сейчас мы этим заниматься не будем. Если программа завершилась успешно, код завершения должен быть равен нулю, поэтому в предложении 9 мы загружаем 0 в регистр AL и вызываем DOS уже знакомой нам командой int 21h. Поскольку выполняемая часть

программы на этом закончилась, можно (и нужно) закрыть сегмент команд, что выполняется с помощью директивы ends (от end segment, конец сегмента), перед которой для наглядности обычно указывается имя закрываемого сегмента, в данном случае сегмента text.

Вслед за сегментом команд описывается сегмент данных. Он, как и сегмент команд, начинается директивой segment, предваряемой произвольным именем нашего сегмента, и заканчивается директивой ends. У нас в качестве данных выступает строка текста. Текстовые строки вводятся в программу с помощью директивы ассемблера db (от define byte, определить байт) и заключаются в апострофы или в кавычки. Для того чтобы в программе можно было обращаться к данным, поля данных, как правило, предваряются именами. В нашем случае таким именем является вполне произвольное обозначение mesg (от message, сообщение), с которого начинается предложение 13.

Выше, в предложении 6, мы через регистр DX передали DOS адрес начала выводимой на экран строки текста. Но как DOS определит, где эта строка закончилась? Хотя нам конец строки в программе отчетливо виден, однако в машинных кодах, из которых состоит выполнимая программа, он никак не отмечен, и DOS, выведя на экран последний символ строки – восклицательный знак, продолжит вывод байтов памяти, расположенных за фразой "Начинаем!". Поэтому DOS следует передать информацию о том, где кончается строка текста. Некоторые функции DOS требуют указания в одном из регистров длины выводимой строки, однако функция 09h работает иначе. Она выводит текст до знака доллара ($), которым мы и завершили нашу фразу.

Сегмент стека, которому мы дали произвольное имя stk, начинается, как и остальные сегменты, оператором segment и заканчивается оператором ends. Назначение стека и особенности его использования будут подробно описаны в одной из последующих статей. Пока ограничимся замечанием, что стек представляет собой отдельный сегмент обычно небольшого объема, в котором просто резервируется определенное количество пустых байтов. Для выделения в программе группы байтов используется конструкция

```
db размер dup (заполнитель)
```

В нашем примере для стека выделено 256 байт, заполненных нулями.

Оператор segment, начинающий сегмент стека, имеет описатель stack. Указание этого обозначения приводит к тому, что при загрузке программы в память регистры процессора, используемые для работы со стеком, инициализируются системой должным образом. Конкретно, сегментный регистр стека SS будет настроен на начало сегмента стека, а указатель стека SP – на его конец (как мы увидим дальше, стек заполняется данными от конца к началу). При отсутствии описателя stack нам пришлось бы выполнять инициализацию регистров, связанных со стеком, вручную.

Последняя строка программы содержит директиву end, которая говорит программе ассемблера, что закончился вообще весь текст программы и больше ничего транслировать не нужно. В качестве операнда этой директивы, как уже отмечалось, обычно указывается точка входа в программу, т. е. адрес первой выполнимой программной строки. В нашем случае это метка begin.

Статья 2. **Подготовка программы к выполнению**

Процесс подготовки и отладки программы на языке ассемблера включает этапы подготовки файла с исходным текстом, его трансляции и компоновки и, наконец, отладки программы с помощью специальной программы интерактивного отладчика.

Подготовка исходного текста программы выполняется с помощью любого текстового редактора. Файл с исходным текстом должен иметь расширение .ASM. При выборе редактора для подготовки исходного текста программы следует иметь в виду, что многие текстовые процессоры (например, Microsoft Word) добавляют в выходной файл служебную информацию о формате (размер страниц, типы используемых шрифтов и др.). Поэтому следует воспользоваться редактором, выводящим в выходной файл "чистый текст", без каких-либо управляющих символов. К таким редакторам относится, например, программа Norton Editor, а также редактор EDIT.COM, входящий в состав операционной системы MS-DOS. Можно воспользоваться и простым редактором, встроенным в оболочку Norton Commander (клавиша F4). Поскольку при интенсивном программировании часто приходится переносить фрагменты текста из одной программы в другую, желательно, чтобы редактор имел средство деления экрана на независимые окна. Таким свойством обладает редактор Norton Editor, позволяющий работать одновременно в двух окнах, что в большинстве случаев вполне достаточно. При работе в операционной среде какой-либо системы программирования, например Borland C, можно воспользоваться редактором, встроенным в эту среду.

Трансляция исходного текста программы состоит в преобразовании предложений исходного языка в коды машинных команд и выполняется с помощью транслятора с языка ассемблера (т. е. с помощью программы ассемблера). Можно воспользоваться макроассемблером корпорации IBM, пакетами TASM корпорации Borland или Microsoft MASM. Трансляторы различных разработчиков имеют некоторые различия, в основном в части описания макросредств. Однако входной язык (т. е. мнемоника машинных команд и других операторов и правила написания предложений ассемблера) для всех ассемблеров одинаков. В результате трансляции образуется объектный файл с расширением .OBJ (рис. 2.1).



Рис. 2.1. Процесс подготовки программы к выполнению

Компоновка объектного файла выполняется с помощью программы компоновщика (редактора связей). Эта программа получила такое название потому, что ее основное назначение – подсоединение к файлу с основной программой файлов с подпрограммами и настройка связей между ними. Однако компоновать необходимо даже простейшие программы, не содержащие подпрограмм. Дело в том, что у компоновщика есть и вторая функция – изменение формата объектного файла и преобразование его в выполнимый файл, который может быть загружен в оперативную память и выполнен. Файл с программой компоновщика обычно имеет имя LINK.EXE, хотя это может быть и не так. Например, компоновщик корпорации Borland назван TLINK.EXE. Компо-

новщик необходимо брать из одного пакета с ассемблером. В результате компоновки образуется загрузочный, или выполнимый, файл с расширением .EXE.

Отладка готовой программы может выполняться разными методами, выбор которых определяется структурой и функциями отлаживаемой программы. Свою специфику отладки имеют, например, резидентные программы, обработчики аппаратных прерываний, драйверы устройств и другие классы программ. В целом наиболее удобно отлаживать программы с помощью какого-либо интерактивного отладчика, который позволяет выполнять отлаживаемую программу по шагам или с точками останова, выводить на экран содержимое регистров и областей памяти, модифицировать (в известных пределах) загруженную в память программу, принудительно изменять содержимое регистров и выполнять другие действия, позволяющие в наглядной и удобной форме контролировать выполнение программы.

При использовании пакета Borland следует взять "турбо-дебаггер" TD.EXE, при трансляции и компоновке программы с помощью пакета Microsoft – отладчик Codeview (файл CV.EXE). Можно воспользоваться и отладчиком DEBUG.EXE, входящим в состав операционной системы MS-DOS, хотя с ним не очень удобно работать, так как эта программа не обеспечивает привычный для сегодняшнего пользователя полноэкранный интерфейс. В настоящей книге будет предполагаться, что читатель выполняет предложенные примеры с помощью пакета TASM (транслятор TASM.EXE, компоновщик TLINK.EXE, отладчик TD.EXE), хотя с тем же успехом можно воспользоваться пакетом MASM или любым другим.

Если файл с исходным текстом программы назван P.ASM, то строка вызова ассемблера может иметь следующий вид:

```
tasm /z /zi /n p, p, p
```

(Еще раз напоминаем, что как в тексте программы на языке ассемблера, так и при вводе с клавиатуры командных строк можно с равным успехом использовать и прописные и строчные буквы.)

Ключ /z разрешает вывод на экран строк исходного текста программы, в которых ассемблер обнаружил ошибки (без этого ключа поиск ошибок пришлось бы всегда проводить по листингу трансляции).

Ключ /zi управляет включением в объектный файл номеров строк исходной программы и другой информации, не требуемой при выполнении программы, но используемой отладчиком.

Ключ /n подавляет вывод в листинг перечня символических обозначений в программе, от чего несколько уменьшается информативность листинга, но существенно сокращается его размер.

Стоящие далее параметры определяют имена файлов: исходного (P.ASM), объектного (P.OBJ) и листинга (P.LST). Расширения имен файлов можно не указывать.

Строка вызова компоновщика может иметь следующий вид:

```
tlink /v /x p, p
```

Ключ /v передает в загрузочный файл символьную информацию, позволяющую отладчику TD выводить на экран полный текст исходной программы, включая метки, комментарии и пр. Стоящие далее параметры обозначают имена модулей: объектного (P.OBJ) и загрузочного (P.EXE). Ключ /x подавляет формирование карты загрузки (файла с листингом компоновки P.MAP), без которого вполне можно обойтись.

Как уже отмечалось, компоновщик создает загрузочный, готовый к выполнению модуль в формате .EXE. Запуск подготовленной программы P.EXE осуществляется командой

```
p.exe
```

или просто

```
p
```

Если программа не работает должным образом, необходимо прибегнуть к помощи интерактивного отладчика. Отладчик пакета TASM запускается командой

```
td p
```

где p (или p.exe) – имя файла с отлаживаемой программой. По умолчанию отладчик загружает файл с расширением .EXE. В процессе работы отладчик использует также файл с исходным модулем P.ASM, поэтому перед отладкой не следует переименовывать ни исходный, ни выполнимый файлы. Правила работы с отладчиком TD и его основные команды будут описаны в статье 4.

Поскольку по мере изучения этой книги вам придется написать и отладить несколько десятков программ, целесообразно создать командный файл, автоматизирующий выполнение однотипных операций трансляции и компоновки. Текст командного файла может быть таким:

```
@echo off
tasm /z /zi /n p,p,p
if errorlevel 1 goto err
tlink /v /x p,p
goto end
:err
echo Ошибка трансляции!
goto fin
:end
echo Конец сеанса
:fin
echo .
```

Приведенный текст составлен в предположении, что путь к программам пакета TASM указан в команде PATH. Если это по каким-либо причинам не так, в командный файл следует включить полную спецификацию файлов ассемблера и компоновщика, например (если весь пакет находится на диске D: в каталоге TASM):

```
d:\tasm\tasm /z /zi /n p,p,p
```

Если трансляция прошла успешно, на экран выводится сообщение "Конец сеанса" и создаются файлы P.OBJ, P.EXE и P.LST; при наличии ошибок в программе на экран будут выведены строки листинга с ошибками, а за ними сообщение "Ошибка трансляции!". Поскольку в приведенном командном файле имя программы указано в явной форме, текущий вариант программы всегда должен иметь одно и то же имя (в приведенном примере – P.ASM). Конечно, можно составить командный файл, воспринимающий текущее имя программы в качестве параметра при его запуске, однако практика показывает, что такая методика замедляет работу и иногда приводит к драматическим ошибкам. Удобнее отлаживать программу всегда под одним и тем же именем, а после отладки записывать в специально созданный каталог архива под уникальным именем (например, под именем, соответствующим номеру примера в книге: 01-01.ASM).

Создайте файл с программой из примера 1.1. Подготовьте программу к выполнению. Запустите программу и убедитесь, что она работает правильно: выводит на экран нужный текст и не нарушает работу компьютера.

Изучите листинг трансляции (файл P.LST), приведенный на рис. 2.2. Обратите внимание на следующие моменты.

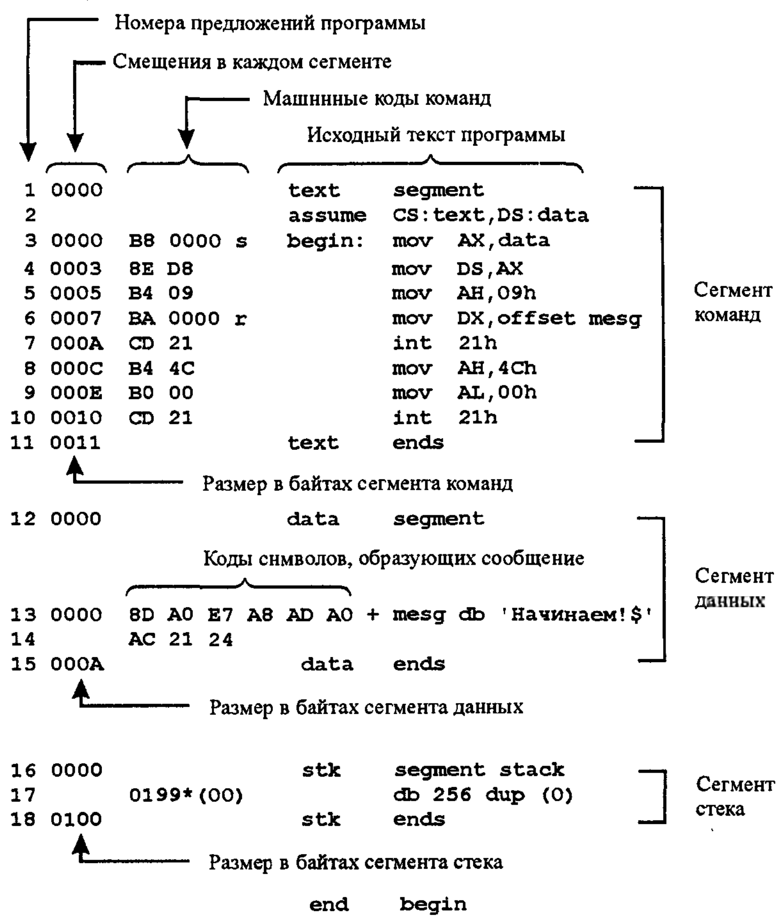


Рис. 2.2. Листинг трансляции примера 1.1

Команды программы имеют различную длину и располагаются в памяти вплотную друг к другу. Так, первая команда mov AX,text, начинающаяся с байта 0000 сегмента, занимает 3 байта. Соответственно, вторая команда начинается с байта 0003. Вторая команда имеет длину 2 байта, поэтому третья команда начинается с байта 0005 и т. д.

Предложения программы с операторами segment, assume, end не транслируются в какие-либо машинные коды и не находят отражения в памяти. Они нужны лишь для передачи транслятору служебной информации.

Транслятор не смог полностью определить код команды mov AX,text. В этой команде в регистр AX засылается адрес сегмента text. Однако этот адрес станет известен лишь в процессе загрузки выполнимого файла программы в память. Поэтому в листин-

ге на месте этого адреса стоят нули. Символ s напоминает нам о том, что в дальнейшем вместо нулей сюда будет подставлен сегментный адрес.

Данные, введенные нами в программу, также оттранслировались: вместо символов текста в загрузочный файл попадут коды ASCII этих символов. Подробнее о кодах ASCII будет рассказано позже.

Из листинга трансляции легко определить размер отдельных составляющих программы и всей программы в целом. В нашем случае длина сегмента команд равна всего 12h=18 байт; длина сегмента данных равна 0Ah=10 байт и в точности совпадает с числом символов в выводимом сообщении (включая знак доллара); длина сегмента стека, как мы и определили в программе, составляет 100h=256 байт.

Выполните пробный прогон программы и убедитесь, что она работает как ожидалось: на экран выводится текст "Начинаем!". Если этого не произойдет, значит, в программе присутствует какая-то скрытая ошибка и программу придется отлаживать с помощью отладчика. Работа с интерактивным отладчиком требует некоторых навыков; статья 4 поможет вам освоить это интересное и полезное инструментальное средство.

Статья 3. **Регистры процессора**

В статье 1 при описании приведенной там программы упоминались регистры процессора, в частности сегментные и общего назначения. Регистры процессора являются важнейшим инструментом программиста (между прочим, не только на языке ассемблера, но и на языках высокого уровня), и необходимо иметь четкое представление о составе регистров процессора, их названиях и применении. В примере 1.1 в явной форме использовались лишь три регистра процессора – AX (и его старшая половина AH), DS и DX; по мере чтения книги вы столкнетесь с примерами использования остальных регистров. Однако для того, чтобы читатель мог получить общее представление об этом важном средстве, ниже дан краткий обзор всей системы регистров процессора. Если в этом обзоре вам встретятся незнакомые понятия или неясные места – не огорчайтесь; в последующих статьях они будут описаны более подробно.

Строго говоря, приведенное ниже описание относится только к процессорам 8086 и 80286, для которых были характерны 16-разрядные регистры. В современных процессорах типа Pentium почти все регистры 32-разрядные, что существенно увеличивает возможности компьютера. Однако младшие половины регистров этих процессоров совпадают и по названиям и по назначению с 16-разрядными регистрами процессора 8086. Поэтому программы, написанные для выполнения под управлением MS-DOS, т. е. для 16-разрядного процессора, прекрасно работают и с 32-разрядным, хотя и не используют все его возможности. Поначалу мы рассмотрим 16-разрядную архитектуру процессора 8086 или, точнее, той части современных процессоров, которая предназначена для использования в программах для системы MS-DOS. Мы будем называть этот гипотетический процессор МП 86. Особенности реальных 32-разрядных процессоров будут описаны позже.

МП 86 содержит двенадцать 16-разрядных программно-адресуемых регистров, которые принято объединять в три группы: регистры данных, регистры-указатели и сегментные регистры. Кроме того, в состав процессора входят счетчик команд и регистр флагов (рис. 3.1). Регистры данных и регистры-указатели часто называют регистрами общего назначения.

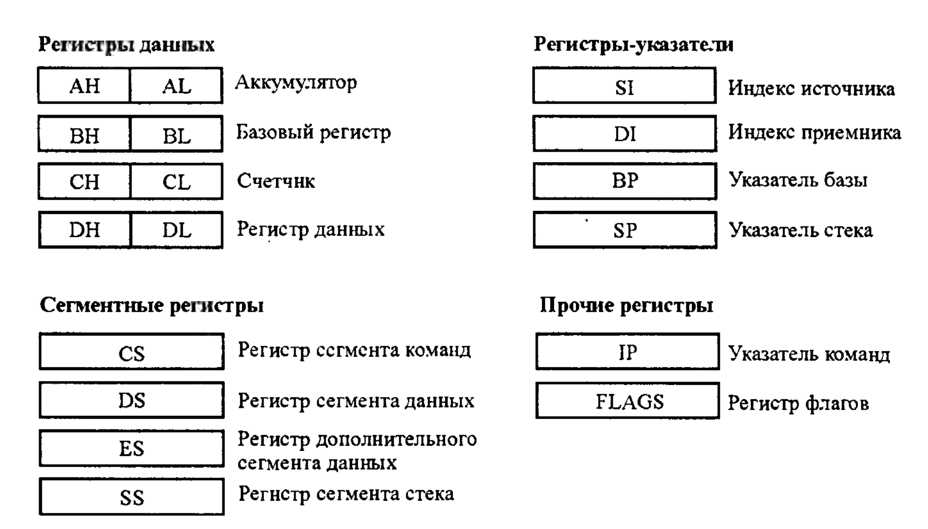


Рис. 3.1. Регистры процессора

В группу регистров данных включаются регистры AX, BX, CX и DX. Программист может использовать их по своему усмотрению для временного хранения любых объектов (данных или адресов) и выполнения над ними требуемых операций. При этом регистры допускают независимое обращение к старшим (AH, BH, CH и DH) и младшим (AL, BL, CL и DL) половинам. Так, команда

```
mov    BL,AH
```

пересылает старший байт регистра AX в младший байт регистра BX, не затрагивая при этом вторых байтов этих регистров. Еще раз отметим, что сначала указывается операнд-приемник, а после запятой – операнд-источник, т. е. команда выполняется как бы справа налево. В качестве средства временного хранения данных все регистры общего назначения (да и все остальные, кроме сегментных и указателя стека) вполне эквивалентны, однако многие команды требуют для своего выполнения использования вполне определенных регистров. Например, команда умножения mul требует, чтобы один из сомножителей был в регистре AX (или AL), а команда организации цикла loop выполняет циклический переход CX раз.

Индексные регистры SI и DI так же, как и регистры данных, могут использоваться произвольным образом. Однако их основное назначение – хранить индексы (смещения) относительно некоторой базы (т. е. начала массива) при выборке операндов из памяти. Адрес базы при этом обычно находится в одном из базовых регистров (BX или BP). Примеры такого рода будут приведены ниже.

Регистр BP служит указателем базы при работе с данными в стековых структурах, о чем будет речь впереди, но может использоваться и произвольным образом в большинстве арифметических и логических операций или просто для временного хранения каких-либо данных.

Последний из регистров-указателей, указатель стека SP, стоит особняком от других в том отношении, что используется исключительно как указатель вершины стека. Стек уже упоминался в статье 1, и будет подробно описан позже.

Регистры SI, DI, BP и SP, в отличие от регистров данных, не допускают побайтовую адресацию.

Четыре сегментных регистра CS, DS, ES и SS хранят начальные адреса сегментов программы и, тем самым, обеспечивают возможность обращения к этим сегментам.

Регистр CS обеспечивает адресацию к сегменту, в котором находятся программные коды, регистры DS и ES – к сегментам с данными (таким образом, в любой точке программа может иметь доступ к двум сегментам данных, основному и дополнительному), а регистр SS – к сегменту стека. Сегментные регистры, естественно, не могут выступать в качестве регистров общего назначения.

Указатель команд IP "следит" за ходом выполнения программы, указывая в каждый момент относительный адрес команды, следующей за исполняемой. Регистр IP программно недоступен (IP – это просто его сокращенное название, а не мнемоническое обозначение, используемое в языке программирования); наращивание адреса в нем выполняет микропроцессор, учитывая при этом длину текущей команды.

Регистр флагов, эквивалентный регистру состояния процессора других вычислительных систем, содержит информацию о текущем состоянии процессора (рис. 3.2). Он включает 6 флагов состояния и 3 бита управления состоянием процессора, которые, впрочем, тоже обычно называются флагами.

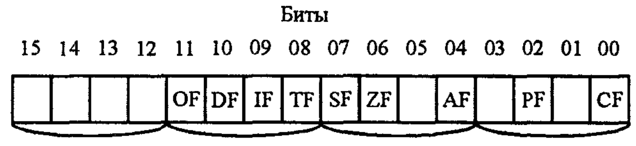


Рис. 3.2. Регистр флагов. Дужками выделены четверки битов

Флаг переноса CF (Carry Flag) индицирует перенос или заем при выполнении арифметических операций, а также (что для прикладного программиста гораздо важнее!) служит индикатором ошибки при обращении к системным функциям.

Флаг паритета PF (Parity Flag) устанавливается в 1, если младшие 8 бит результата операции содержат четное число двоичных единиц.

Флаг вспомогательного переноса AF (Auxiliary Flag) используется в операциях над упакованными двоично-десятичными числами. Он индицирует перенос в старшую тетраду (четверку битов) или заем из старшей тетрады.

Флаг нуля ZF (Zero Flag) устанавливается в 1, если результат операции равен нулю.

Флаг знака SF (Sign Flag) показывает знак результата операции, устанавливаясь в 1 при отрицательном результате.

Флаг переполнения OF (Overflow Flag) фиксирует переполнение, т. е. выход результата операции за пределы допустимого для данного процессора диапазона значений.

Флаги состояния автоматически устанавливаются процессором после выполнения каждой команды. Так, если в регистре AX содержится число 1, то после выполнения команды декремента (уменьшения на единицу)

```
dec AX
```

содержимое AX станет равно нулю и процессор сразу отметит этот факт, установив в регистре флагов бит ZF (флаг нуля). Если попытаться сложить два больших числа, например 58 000 и 61 000, то установится флаг переноса CF, так как число 119 000, получающееся в результате сложения, должно занять больше двоичных разрядов, чем помещается в регистрах или ячейках памяти, и возникает "перенос" старшего бита этого числа в бит CF регистра флагов.

Индицирующие флаги процессора дают возможность проанализировать, если это нужно, результат последней операции и осуществить "разветвление" программы: например, в случае нулевого результата перейти на выполнение одного фрагмента программы, а в случае ненулевого – на выполнение другого. Такие разветвления осуществляются с помощью команд условных переходов, которые в процессе своего выполнения анализируют состояние регистра флагов. Так, команда

```
jz zero
```

осуществляет переход на метку zero, если результат выполнения предыдущей команды окажется равен нулю (т. е. флаг ZF установлен), а команда

```
jnc okey
```

выполнит переход на метку okey, если предыдущая команда сбросила флаг переноса CF (или оставила его в сброшенном состоянии).

Управляющий флаг трассировки TF (Trace Flag) используется в отладчиках для осуществления пошагового выполнения программы. Если TF=1, то после выполнения каждой команды процессор реализует процедуру прерывания 1 (через вектор прерывания с номером 1).

Управляющий флаг разрешения прерываний IF (Interrupt Flag) разрешает (если равен единице) или запрещает (если равен нулю) процессору реагировать на прерывания от внешних устройств.

Управляющий флаг направления DF (Direction Flag) используется особой группой команд, предназначенных для обработки строк. Если DF=0, строка обрабатывается в прямом направлении, от меньших адресов к большим; если DF=1, обработка строки идет в обратном направлении.

Таким образом, в отличие от битов состояния, управляющие флаги устанавливает или сбрасывает программист, если он хочет изменить настройку системы (например, запретить на какое-то время аппаратные прерывания или изменить направление обработки строк).

Вернемся к примеру 1.1. Для того чтобы инициализировать сегментный регистр DS сегментным адресом data нашего сегмента данных, значение data загружается сначала в регистр общего назначения AX, а из него – в сегментный регистр DS. В принципе в качестве "перевалочного пункта" вместо регистра AX можно взять любой другой (например, BX или SI), однако некоторым трансляторам это может не понравиться, так что лучше все-таки использовать AX.

В предложении 5 в регистр AH заносится номер функции DOS, реализующей вывод на экран строки текста. DOS, получив управление с помощью команды int 21h, определяет номер требуемой функции именно по содержимому регистра AH, поэтому никаким другим регистром здесь воспользоваться нельзя. Функция DOS вывода строки извлекает адрес выводимой строки из регистра DX, поэтому в предложении 6 использование регистра DX также предопределено. В действительности дело обстоит сложнее. Функция 09h предполагает, что строка с выводимым текстом находится в сегменте, на который указывает вполне определенный сегментный регистр, именно регистр DS. Поэтому перед вызовом функции 09h необходимо настроить этот регистр, что мы и сделали в предыдущих предложениях программы. Сведения о том, какие регистры требуется настроить для выполнения той или иной функции DOS, можно почерпнуть из справочника по функциям DOS (см. Приложение 4), без которого, таким

образом, практически невозможно писать программы с обращением к системным средствам.

В предложениях 8...10 осуществляется вызов системной функции 4Ch, служащей для завершения текущей программы. По-прежнему номер функции заносится в регистр AH; кроме этого, в AL помещается код завершения программы, который в нашем примере равен нулю. Можно было поступить проще, сразу занеся в регистр AX и номер функции и код завершения:

```
mov    AX,4C00h
```

Это предложение полностью эквивалентно варианту, использованному в примере 1.1, но выглядит компактнее, хотя, возможно, не так наглядно. Необходимо понимать, что, если мы заносим в 2-байтовый регистр 16-битовое число, старшая половина числа поступает в старшую часть регистра, а младшая – в младшую (рис. 3.3).

mov AX,1234h

12h	34h
AH	AL

Рис. 3.3. Расположение байтов 16-битового числа в 16-разрядном регистре

Заметим, что программист использует в программе те или иные регистры процессора лишь по мере необходимости. Так, в примере 1.1 не использовались регистры CX, BP, ES и др.

Статья 4. **Интерактивный отладчик TD**

Часто бывает так, что программа, успешно пройдя этапы трансляции и компоновки, все же работает не так, как ожидалось или вообще не работает. Это значит, что формально, с точки зрения правил языка программирования, программа написана правильно (в ней нет синтаксических ошибок), однако алгоритм ее в чем-то неверен. Для отладки такой программы следует воспользоваться услугами интерактивного отладчика. Интерактивным он называется потому, что вся работа с ним осуществляется в непрерывном диалоге с пользователем.

Познакомимся с отладчиком TD.EXE из пакета TASM, воспользовавшись программой из примера 1.1.

Как уже отмечалось, для полного использования возможностей отладчика следует при трансляции программы указать в числе других ключ /zi, а при компоновке – ключ /co:

```
tasm /z /zi /n p,p,p
tlink /v p,p
```

Кроме того, следует убедиться, что в вашем рабочем каталоге имеется и загрузочный (P.EXE) и исходный (P.ASM) файл, поскольку отладчик в своей работе использует оба этих файла. Вызовем отладчик командой

```
td p
```

На экране появится кадр отладчика, в котором видны два окна – окно Module с исходным текстом отлаживаемой программы и окно Watches для наблюдения за ходом изменения заданных переменных в процессе выполнения программы (рис. 4.1). Окно Watches нам не понадобится, и его можно убрать, щелкнув мышью по маленькому

квадратику в левом верхнем углу окна или введя команду Alt+F3, предварительно сделав это окно активным. Переключение (по кругу) между окнами осуществляется клавишей F6.

```
Ë File Edit View Run Breakpoints Data Options Window Help       READY
╔[■]═Module: p File: p.asm 3═══════════════════════════════════1═[↑][↓]╗
║ text  segment                                                        ▲
║    assume  cs:text,ds:data                                           ■
▶ begin:   mov   AX,data
•    mov   DS,AX
•    mov   AH,09h
•    mov   DX,offset mesg
•    int   21h
•    mov   AH,4Ch
•    mov   AL,0
•    int   21h
║ text  ends
║ data  segment
║ mesg  db "Начинаем!$"
║ data  ends
║ stk   segment stack
║    db 256 dup (0)
║    ends
║    end   begin                                                       ▼
╚◄■████████████████████████████████████████████████████████████████►┘
┌───Watches──────────────────────────────────────────────────2───────┐
│                                                                    │
└────────────────────────────────────────────────────────────────────┘
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Рис. 4.1. Начальный кадр отладчика

Верхняя строка кадра отладчика представляет собой главное меню. Для перехода в меню необходимо нажать клавишу Alt и первую (выделенную цветом) букву требуемого пункта. Для выбора затем конкретного действия надо с помощью клавиш со стрелками ↑ и ↓ выделить нужный пункт и нажать клавишу Enter.

В нижней строке отладчика приведены его основные команды, вызов которых осуществляется нажатием на функциональные клавиши F1...F10. В действительности команд гораздо больше; некоторые из них можно реализовать только с помощью пунктов главного меню, другие вызываются комбинациями функциональных и управляющих (Alt, Ctrl или Shift) клавиш.

Рассмотрим типичные действия, к которым приходится прибегать по ходу отладки программы.

После вызова отладчика и загрузки в память отлаживаемой программы ее первое предложение помечается значком ▶; там же устанавливается и значок подчеркивания _, который играет в отладчике роль курсора. По мере выполнения программных предложений значок ▶ будет перемещаться по тексту программы, всегда указывая на очередное (еще не выполненное) предложение; значок _ можно перемещать с помощью клавиш со стрелками.

Нажав одну из клавиш F8 или F7, мы выполним одно предложение программы. Различие этих команд весьма существенно: команда F7 (trace, трассировка) позволяет войти внутрь вызываемых подпрограмм, а также выполнять циклы шаг за шагом. Команда F8 (step, шаг), наоборот, выполняет подпрограммы и циклы как одно неразрыв-

ное действие, что заметно ускоряет пошаговую отладку программы, если мы, например, уверены, что вызываемая подпрограмма выполняется правильно.

Можно выполнить сразу и целый фрагмент программы, т. е. несколько предложений. Для этого надо поместить курсор _ перед тем предложением, на котором требуется сделать остановку (или на любой символ внутри него), и нажать клавишу F4 (here, сюда). Выполнятся все строки программы до той, на которой установлен курсор; значок ▶ переместится на эту строку. Далее можно опять выполнять программу построчно, нажимая на клавишу F8, или, установив в требуемом месте курсор, выполнить следующий фрагмент, нажав F4.

Для повторного выполнения программы ее следует "рестартовать". Для этого надо выбрать в главном меню пункт Run➤Program reset или просто нажать Ctrl+F2.

Важнейшим элементом отладки программы является наблюдение значений тех или иных полей данных, особенно тех, которые заполняются программой динамически, т. е. по ходу ее выполнения. Для того чтобы вывести на экран содержимое поля данных, надо поместить курсор на имя этого поля (например, mesg в нашем примере) и выбрать пункт меню Data➤Inspect. В появившемся окне ввода переменной (рис. 4.2) можно скорректировать имя интересующего нас поля данных или ввести новое; если имя правильное, достаточно нажать клавишу Enter.

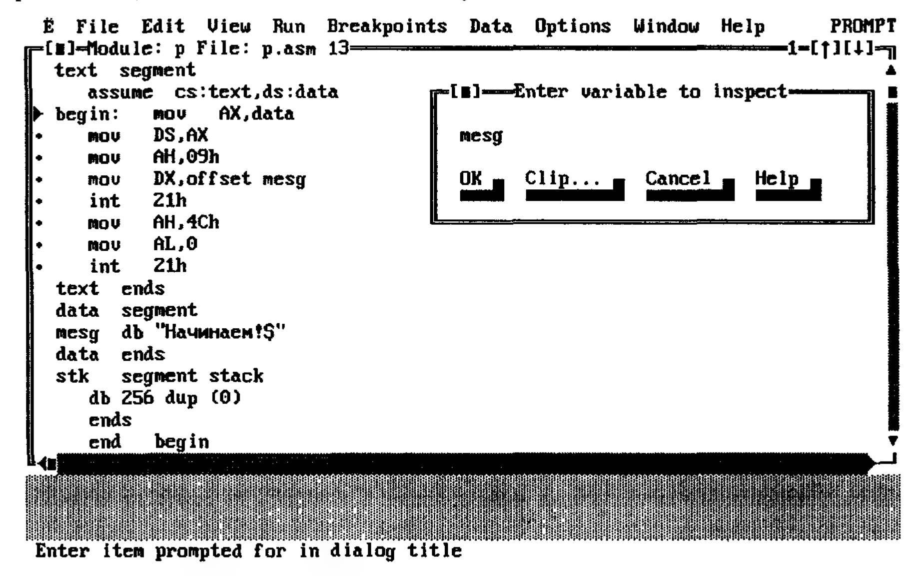


Рис. 4.2. Внутреннее окно отладчика для ввода имени поля данных

В кадр отладчика будет выведено окно с характеристиками и содержимым указанной переменной (рис. 4.3). Отладчик сообщает, что переменная mesg хранится в памяти по адресу 1D18:000, т. е. имеет сегментный адрес 1D18h и смещение 0000h, и описана как последовательность из 10 байт. Тут же приводятся значения всех байтов нашей строки, включая их начертание на экране, а также десятичное и 16-ричное представление.

```
 Ё  File  Edit  View  Run  Breakpoints  Data  Options  Window  Help       READY
┌──Module: p File: p.asm 13──────────────────────────────────────────1──┐
│  text  segment
│     assume  cs:text,ds:data                     ╔[■]=Inspecting mesg=2=[↑][↓]╗
│► begin:   mov    AX,data                        ║@1D18:0000                   ║
│•    mov    DS,AX                                ║[0]             'Н' 141 (8Dh)║
│•    mov    AH,09h                               ║[1]             'а' 160 (A0h)║
│•    mov    DX,offset mesg                       ║[2]             'ч' 231 (E7h)║
│•    int    21h                                  ║[3]             'и' 168 (A8h)║
│•    mov    AH,4Ch                               ║[4]             'н' 173 (ADh)║
│•    mov    AL,0                                 ║[5]             'а' 160 (A0h)║
│•    int    21h                                  ║[6]             'е' 165 (A5h)║
│  text  ends                                     ║[7]             'м' 172 (ACh)║
│  data  segment                                  ║[8]              '!' 33 (21h)║
│  mesg  db "Начинаем!$"                          ║[9]              '$' 36 (24h)║
│  data  ends                                     ║                             ║
│  stk   segment stack                            ║◄■▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓► ║
│     db 256 dup (0)                              ║byte [10]                    ║
│     ends                                        ╚═════════════════════════════╝
│     end   begin
└───────────────────────────────────────────────────────────────────────┘

F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Рис. 4.3. Вывод на экран содержимого поля данных

В окне Inspecting можно изменить значение отображаемого поля данных. Для этого надо, сделав это окно активным и поместив курсор на отображение конкретного элемента нашего символьного массива, например элемента с индексом 8 (знак "!"), ввести команду Alt+F10. Эта команда для любого активного окна открывает его внутреннее меню с дополнительными возможностями. В данном случае внутреннее меню будет иметь вид, показанный на рис. 4.4.

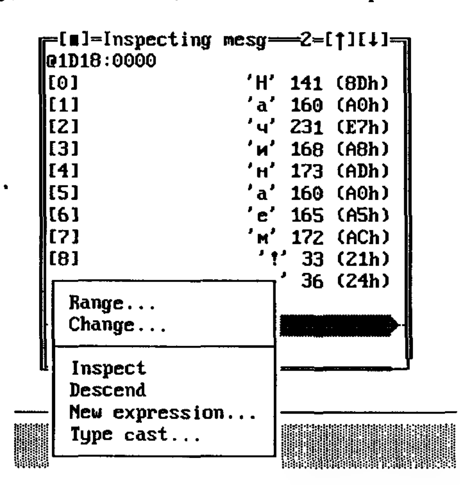


Рис. 4.4. Фрагмент кадра отладчика с внутренним окном для окна Inspecting

Нас будет интересовать пункт Change (изменение). Выбрав этот пункт, мы получим окно, в котором можно ввести требуемое значение изменяемого данного. На рис. 4.5 показано это окно с введенным символом '>', которым будет заменен восклицательный знак. Можно было вместо символа в одинарных кавычках ввести его 16-ричный код ASCII, если он известен (число 3E для знака >). Допустим ввод и десятичного кода, если завершить его буквой d (62d).

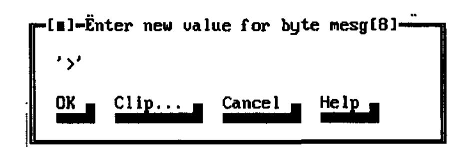


Рис. 4.5. Окно изменения выбранного элемента

Если теперь, не выходя из отладчика, выполнить программу до конца, на экран будет выведена фраза

Начинаем>

Для того чтобы, находясь в отладчике, увидеть результат работы программы, надо ввести команду Alt+F5 (или выбрать пункт Window➤User screen главного меню). Возврат в кадр отладчика осуществляется нажатием любой клавиши.

Необходимо отдавать себе отчет в том, что любые изменения, вносимые в текст программы в процессе работы с отладчиком, будут действовать только до конца данного сеанса (даже, точнее говоря, до рестарта программы). Отладчик изменяет не файл P.EXE, хранящийся на диске, а только его копию в памяти. После завершения сеанса работы с отладчиком все, что было загружено в память, исчезает вместе с внесенными нами изменениями.

Начальное окно отладчика дает недостаточно информации для серьезной работы с программой. При отладке программы на уровне языка ассемблера необходимо контролировать все регистры процессора, включая регистр флагов, а также во многих случаях поля данных вне программы (например, векторы прерываний или системные таблицы). Для этого надо командой Alt+V, C (пункт главного меню View➤CPU) открыть "окно процессора" (рис. 4.6).

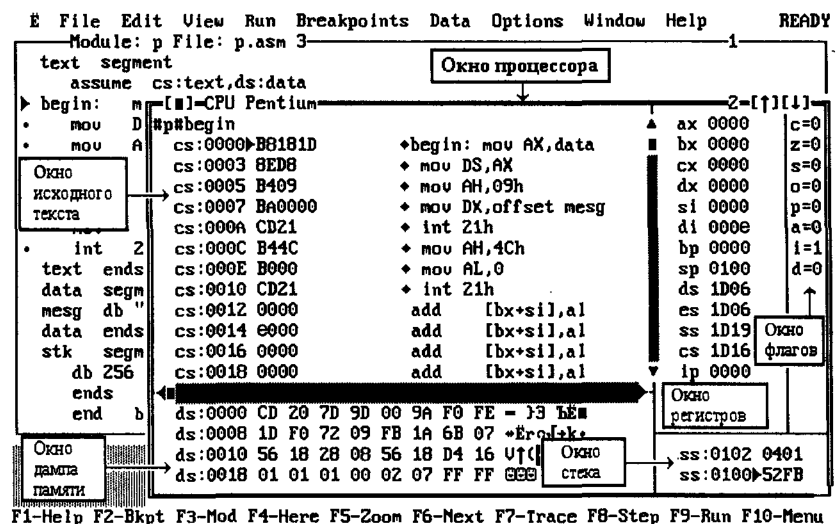


Рис. 4.6. Окно процессора с внутренними окнами

Окно процессора состоит, в свою очередь, из пяти внутренних окон для наблюдения текста программы на языке ассемблера и в машинных кодах, регистров процессора, флагов, стека и содержимого памяти. С помощью этих окон можно полностью контролировать работу процессора при выполнении отлаживаемой программы. Для того

чтобы можно было работать с конкретным окном, надо сделать его активным, щелкнув по нему мышью. Переходить из окна в окно можно также (по кругу), нажимая клавишу Tab. Для управления ходом программы используются функциональные клавиши, перечисленные в нижней строке кадра (F7 или F8 для пошагового выполнения, F4 для выполнения до курсора и т. д.). Курсор во всех внутренних окнах окна процессора выглядит в виде синей ленточки. Добавим еще, что, щелкнув мышью по значку стрелки ↑ в правом верхнем углу окна процессора, можно увеличить это окно до размеров кадра отладчика.

Продемонстрируем некоторые возможности окна процессора. По ходу пошагового выполнения программы можно изменять содержимое регистров. Это дает возможность исправлять обнаруженные ошибки (если выяснилось, что в какой-то строке программы заполняется не тот регистр или не тем содержимым), а также динамически модифицировать программу с целью изучения ее работы.

Выполните программу до первой команды int 21h (предложение 7) и просмотрите содержимое регистров процессора. Вы увидите, что в старшей половине регистра AX находится число 09h – номер вызываемой функции DOS. Младшая половина регистра AX заполнена "мусором" – остатком от выполнения последней операции с регистром AX. В регистре DX будет 0000h – относительный адрес первого байта строки mesg в сегменте команд. Изменим этот относительный адрес. Для этого надо перейти в окно регистров, поместить курсор на строку, отображающую содержимое регистра DX, и ввести команду Alt+F10, открывающую внутреннее меню окна регистров (рис. 4.7).

```
Increment
Decrement
Zero
Change...
Registers 32-bit    No
```

Рис. 4.7. Внутреннее меню окна регистров

Как видно из рис. 4.7, меню окна регистров предоставляет возможность выполнить увеличение содержимого регистра на 1 (Increment), уменьшить его на 1 (Decrement), обнулить (Zero) и заменить на любое заданное значение (Change). Выбрав пункт Change, занесем в регистр DX число 5 (рис. 4.8).

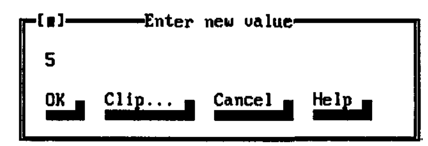


Рис. 4.8. Окно задания регистру произвольного значения

Теперь, если выполнить очередную команду (int 21h), DOS выведет на экран строку, начало которой расположено в байте 5 сегмента данных. В нашей фразе "Начинаем!" байт 5 приходится на вторую букву "а" (нумерация байтов в строке, естественно, начинается с нуля). В результате на экран будет выведена строка

```
аем!
```

а читатель, возможно, немного глубже разберется в том, как в DOS передаются параметры вызываемых нами функций DOS.

Еще одной полезной операцией является вывод содержимого области памяти, начиная с заданного адреса (дамп памяти). Таким образом можно изучать не только поля данных программы, но и содержимое вообще любых участков памяти компьютера,

в частности системных таблиц. Посмотрим, например, как выглядит в памяти блок окружения программы.

Любая программа, загруженная в память, состоит из двух отдельных блоков: собственно программы и ее окружения, которое располагается обычно перед программой, хотя не обязательно вплотную к ней (рис. 4.9).

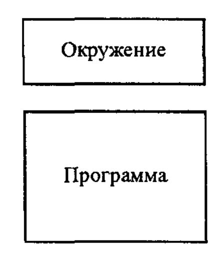


Рис. 4.9. Программа и ее окружение

Окружение представляет собой область памяти, в которой в виде символьных строк записаны значения переменных, называемых переменными окружения, например

```
PROMPT= $p$g
PATH=C:\DOS; C:\TOOLS; C:\NC;
```

Здесь PROMPT и PATH – переменные окружения, а справа от знака равенства указаны их конкретные значения, которое могут быть и другими. Переменная PROMPT определяет форму системного запроса, выводимого командным процессором на экран после завершения любой текущей программы, а переменная PATH – пути к программным файлам, которые будут вызываться на выполнение просто по именам, без указания полной спецификации.

Пользователь может включить в окружение строки определения дополнительных переменных с помощью команды SET. Часто в качестве значений таких переменных указываются пути к каталогам со вспомогательными файлами или ключи, задающие режим работы различных программ.

При загрузке прикладной программы содержимое начального окружения копируется в создаваемое окружение прикладной программы, которая, таким образом, имеет доступ как к системным переменным (которые, скорее всего, ей не нужны), так и к переменным, включенным в окружение пользователем и адресованным именно ей.

Сегментный адрес окружения загружаемой программы записывается системой в ячейку со смещением 2Ch от начала программы. Необходимо заметить (этот вопрос будет подробно рассмотрен в следующей статье), что система, загружая программу в память, пристраивает перед ней специальную системную область, называемую префиксом программы (program segment prefix, PSP), заполняя его данными, которые нужны системе для правильного взаимодействия с программой. Начало PSP видно в нижней части рис. 4.6, в окне дампа памяти. В префиксе программы и находится адрес окружения. Вооружившись этими знаниями, найдем в памяти окружение нашей программы.

Перейдя в окно дампа памяти, введем команду Alt+F10, открывающую внутреннее меню окна дампа (рис. 4.10).

Выбрав пункт Goto (перейти), в открывшемся окне задания адреса перехода укажем смещение 2C. В окно дампа будет выведено содержимое PSP начиная с адреса 2Ch. Однако анализировать его неудобно, так как окно дампа отображает последовательные байты, а искомый адрес занимает слово.

Изменим формат дампа, еще раз введя команду Alt+F10 и выбрав во внутреннем меню пункт Display as (показать в виде). На экран будет выведено еще одно меню (рис. 4.11), в котором можно выбрать требуемый вид представления информации в окне дампа (в виде байтов, слов, двойных слов и т. д.).

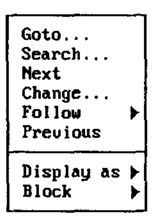


Рис. 4.10. Внутреннее меню окна дампа

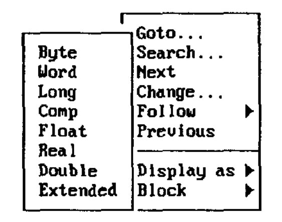


Рис. 4.11. Внутреннее меню окна дампа и подменю пункта Display as

Выберем режим отображения содержимого памяти в виде слов (Words). Мы увидим, что по адресу 2Ch расположено число 1124h (рис. 4.12). Это и есть сегментный адрес окружения текущей программы, характеризующий расположение блока окружения в физической памяти компьютера. Его конкретное значение в известной степени случайно, так как зависит и от размера загруженной в память части DOS, и от наличия резидентных программ (русификатора и др.). У читателя, повторяющего рассматриваемый пример, наверняка получится другой адрес.

```
ds:002C 1124 0E52 1C14 0014
ds:0034 0018 1D06 FFFF FFFF
ds:003C 0000 0000 1606 0000
ds:0044 0000 0000 0000 0000
```

Рис. 4.12. Дамп участка PSP в виде слов

Теперь можно осуществить переход в окне дампа к началу окружения. Сначала, однако, следует опять изменить настройку окна дампа, чтобы получить побайтовый вывод (команды Alt+F10 ➢ Display as ➢ Byte). Опять введем команду Alt+F10 и, выбрав пункт Goto, укажем адрес перехода в виде 1124:0, чтобы посмотреть текст окружения с самого начала отведенного ему сегмента памяти, т. е. со смещения, равного нулю (рис. 4.13).

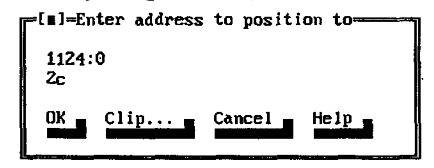


Рис. 4.13. Задание адреса перехода в виде сегмент:смещение

В окно дампа будет выведено начала сегмента окружения (рис. 4.14), где видны переменные окружения CONFIG и COMSPEC. Следует заметить, что фактический состав окружения в сильной степени зависит от конфигурации и настроек компьютера, да и просто от привычек пользователя, так что на каждой машине рис. 4.14 будет выглядеть по-своему.

```
1124:0000 43 4F 4E 46 49 47 3D 66 CONFIG=f
1124:0008 75 6C 6C 5F 63 6F 6E 66 ull_conf
1124:0010 69 67 00 43 4F 4D 53 50 ig COMSP
1124:0018 45 43 3D 43 3A 5C 44 4F EC=C:\DO
```

Рис. 4.14. Внутреннее окно дампа окна процессора с началом текста окружения

К сожалению, в окне процессора отладчика TD нельзя изменять взаимный размер внутренних окон и окно дампа всегда оказывается небольшим. Если мы хотим наблюдать сравнительно большой участок памяти, лучше организовать дамп памяти по-иному.

Выберем пункт главного меню View➢Dump. На экран будет выведено окно дампа, размер которого можно изменять. С помощью той же команды Alt+F10 ➢ Goto вводом адреса 1124:0 перейдем к началу окружения и растянем окно дампа так, чтобы можно было наблюдать весь текст окружения (рис. 4.15).

```
[■]-Dump                                                  4-[↑][↓]
1124:0000 43 4F 4E 46 49 47 3D 66 75 6C 6C 5F 63 6F 6E 66 CONFIG=full_conf
1124:0010 69 67 00 43 4F 4D 53 50 45 43 3D 43 3A 5C 44 4F ig COMSPEC=C:\DO
1124:0020 53 5C 43 4F 4D 4D 41 4E 44 2E 43 4F 4D 00 50 52 S\COMMAND.COM PR
1124:0030 4F 4D 50 54 3D 24 65 5B 33 33 3B 31 6D 24 70 24 OMPT=$e[33;1m$p$
1124:0040 67 24 65 5B 30 6D 00 50 41 54 48 3D 43 3A 5C 57 g$e[0m PATH=C:\W
1124:0050 49 4E 44 4F 57 53 3B 43 3A 5C 3B 43 3A 5C 44 4F INDOWS;C:\;C:\DO
1124:0060 53 3B 43 3A 5C 4E 43 35 3B 43 3A 5C 54 4F 4F 4C S;C:\NC5;C:\TOOL
1124:0070 53 3B 43 3A 5C 4D 41 53 4D 3B 46 3A 5C 57 4F 52 S;C:\MASM;F:\WOR
1124:0080 44 35 3B 43 3A 5C 42 43 34 35 5C 42 49 4E 3B 43 D5;C:\BC45\BIN;C
1124:0090 3A 5C 4D 4F 55 53 45 00 54 45 4D 50 3D 43 3A 5C :\MOUSE TEMP=C:\
1124:00A0 74 65 6D 70 00 54 4D 50 3D 43 3A 5C 74 65 6D 70 temp TMP=C:\temp
1124:00B0 00 42 4C 41 53 54 45 52 3D 41 32 32 30 20 49 37  BLASTER=A220 I7
1124:00C0 20 44 31 20 54 34 00 00 01 00 46 3A 5C 43 55 52  D1 T4  ☺ F:\CUR
1124:00D0 52 45 4E 54 5C 50 2E 45 58 45 00 20 44 65 63 20 RENT\P.EXE  Dec
```

Рис. 4.15. Большое окно дампа памяти с полным текстом окружения

Для изменения размера окна предусмотрено несколько приемов. Клавиша F5 увеличивает активное окно до размеров экрана (повторное нажатие F5 сжимает окно). Введя команду Ctrl+F5, можно затем с помощью клавиш со стрелками →, ↓, ← и ↑ перемещать окно по экрану, а используя клавиши со стрелками → и ↓ при нажатой клавише Shift, изменять размер окна. Наконец, можно растянуть или сжать окно, потянув правый нижний угол окна мышью при нажатой левой клавише. Перемещение окна с помощью мыши осуществляется перетаскиванием его за любое место внешней рамки.

В левой части кадра дампа (см. рис. 4.15) показаны коды (в 16-ричном представлении), содержащиеся в последовательных байтах памяти. В правой части кадра эти коды преобразованы в соответствующие им изображения символов. Рассмотрев дамп окружения, можно заметить, что отдельные строки вида

```
Переменная_окружения = значение
```

разделяются двоичными нулями. Это обстоятельство необходимо иметь в виду при программном анализе окружения с целью поиска в нем значения заданной переменной. В окружении можно найти упоминавшиеся выше переменные PROMPT и PATH; в конце окружения хранится полная спецификация выполняемой программы (F:\CURRENT\P.EXE в данном случае), что позволяет, просматривая память компьютера, определить имена всех загруженных программ. Именно так работает, например, программа DOS MEM, с помощью которой можно получить текущее содержимое памяти.

Статья 5. **Сегментная адресация и сегментная структура программ**

Почему программа должна обязательно состоять из сегментов? Причина этого кроется в архитектурных особенностях микропроцессоров корпорации Intel, которые нам придется здесь коротко рассмотреть. Важнейшей характеристикой любого процессора является разрядность его внутренних регистров, а также внешних шин адресов и данных. МП 86 имеет 16-разрядную внутреннюю архитектуру и такой же разрядности шину данных. Таким образом, максимальное целое число (данное или адрес), с ко-

торым может работать микропроцессор, составляет $2^{16} - 1 = 65\,535$ (64 K–1). Однако адресная шина МП 86 содержит 20 линий, что соответствует адресному пространству 2^{20}=1 Мбайт. Для того чтобы с помощью 16-разрядных адресов можно было обращаться в любую точку 20-разрядного адресного пространства, в процессоре предусмотрена сегментная адресация памяти, реализуемая с помощью четырех сегментных регистров.

Суть сегментной адресации заключается в следующем. Физический 20-разрядный адрес любой ячейки памяти вычисляется процессором путем сложения 20-разрядного начального адреса сегмента памяти, в котором располагается эта ячейка, с 16-разрядным смещением к ней (в байтах) от начала сегмента (рис. 5.1). Начальный адрес сегмента без четырех младших бит, т. е. деленный на 16, хранится в одном из сегментных регистров. Эта величина называется сегментным адресом. Каждый раз при загрузке в сегментный регистр сегментного адреса процессор автоматически умножает его на 10h=16 и полученный таким образом базовый адрес сегмента сохраняет в одном из своих внутренних регистров. При необходимости обратиться к той или иной ячейке памяти процессор прибавляет к этому базовому адресу смещение ячейки, в результате чего образуется физический адрес ячейки в памяти. Умножение 16-разрядного сегментного адреса – 64 Кбайт на 16 увеличивает диапазон адресуемых ячеек до величины 1 Мбайт.

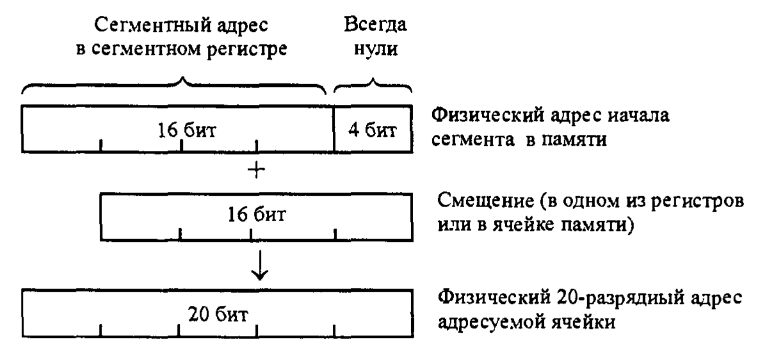


Рис. 5.1. Образование физического адреса из сегментного адреса и смещения

Современные 32-разрядные процессоры Intel, в частности процессоры Pentium, имеют 32-разрядную адресную шину, что соответствует адресному пространству 2^{32} = = 4 Гбайт. Однако описанный выше способ сегментной адресации памяти не позволяет выйти за пределы 1 Мбайт. Для преодоления этого ограничения в 32-разрядных процессорах используются два режима работы: реальный и защищенный. В реальном режиме процессор функционирует фактически так же, как МП 86 с повышенным быстродействием, и может обращаться лишь к 1 Мбайт адресного пространства. Оставшаяся память, даже если она установлена на компьютере, использоваться не может.

В защищенном режимс по-прежнему используются сегменты и смещения в них, однако начальные адреса сегментов не вычисляются путем умножения на 16 содержимого сегментных регистров, а извлекаются из таблиц сегментных дескрипторов, индексируемых с помощью тех же сегментных регистров. Каждый сегментный дескриптор занимает 8 байт, из которых 4 байта (32 бита) отводятся под сегментный адрес. Тем самым обеспечивается полное использование 32-разрядного адресного пространства. В этом случае процессор позволяет адресовать до 2^{32} = 4 Гбайт физической памя-

ти. Программирование защищенного режима будет рассмотрено в 6-м разделе этой книги.

Итак, в МП 86 обращение к любым участкам памяти осуществляется исключительно посредством сегментов – логических образований, накладываемых на требуемые участки физического адресного пространства. Размер сегмента должен находиться в пределах 0 байт ... 64 Кбайт (допустимы и иногда используются сегменты нулевой длины). Начальный адрес сегмента, деленный на 16, т. е. без младшей 16-ричной цифры, заносится (как правило, программистом с помощью соответствующих предложений программы) в один из сегментных регистров. Смещение же адресуемой ячейки указывается тем или иным образом в команде. Процесс адресации памяти проиллюстрирован на рис. 5.2 на примере конкретной команды inc mem1. Если предположить, что ячейка mem1 находится в байтах 4-5 сегмента данных, то полный код команды inc mem1 займет 4 байта и составит такую последовательность 16-ричных чисел:

```
FF 06 04 00
```

В этой последовательности первые 2 байта представляют собой код операции (выполнить инкремент над словом памяти), а вторые два – смещение в сегменте данных к адресуемой ячейке. На рис. 5.2 код операции и смещение изображены в виде двух последовательных слов памяти.

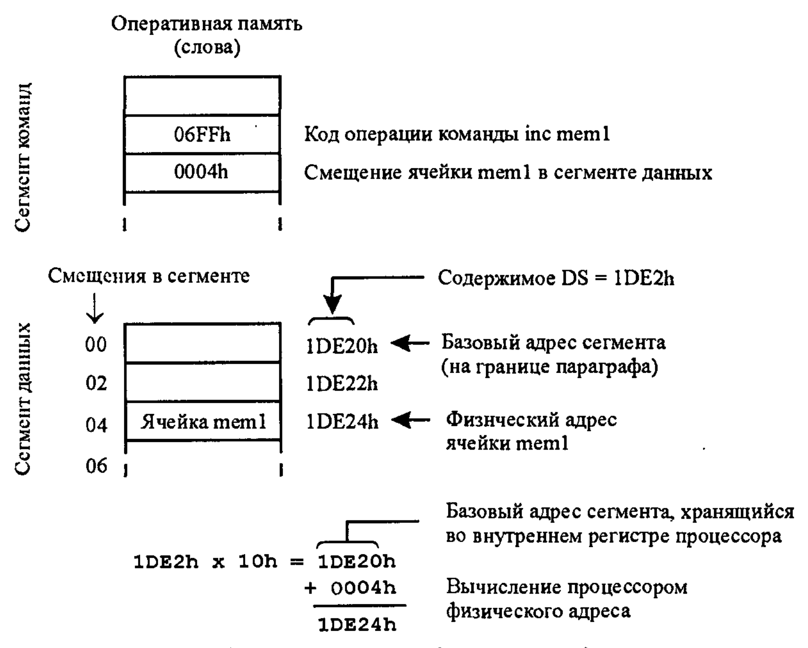


Рис. 5.2. Формирование физического адреса

Поскольку младшая 16-ричная цифра базового адреса сегмента оказывается равной нулю, сегмент всегда начинается с адреса, кратного 16, т. е. на границе 16-байтового блока памяти (параграфа). Следует помнить, что сегментный адрес в 16 раз меньше соответствующего ему физического адреса памяти.

Наличие в микропроцессоре четырех сегментных регистров определяет структуру программы. В типичной, не слишком сложной программе имеются сегмент команд,

сегмент данных и сегмент стека, которые адресуются с помощью сегментных регистров CS, DS и SS соответственно. Дополнительный сегментный регистр ES часто используется для обращения к полям данных, не входящим в программу, например к видеопамяти или системным ячейкам. Однако при необходимости его можно настроить и на один из сегментов программы. В частности, если программа работает с большим объемом данных, для них можно предусмотреть два сегмента и обращаться к одному из них через регистр DS, а к другому – через ES.

Язык ассемблера имеет, в частности, то преимущество, что программа, загруженная в память, полностью повторяет по своей структуре и порядку элементов исходный текст, написанный программистом. Так, если в сегменте данных последовательно описаны некоторые данные, они будут расположены в памяти в точности в том же порядке. Сами сегменты также размещаются в памяти в том порядке, в каком они следуют в исходном тексте. В примере 1.1 мы описали в программе сначала сегмент команд, затем сегмент данных и, наконец, сегмент стека. Удостоверимся в том, что после загрузки программы в память порядок сегментов не изменится и заодно посмотрим, какие у них будут сегментные адреса.

Запустим отладчик с примером 1.1 и выбором пункта главного меню View➢Registers (или командой Alt+V ➢ R) и выведем на экран окно регистров (рис. 5.3).

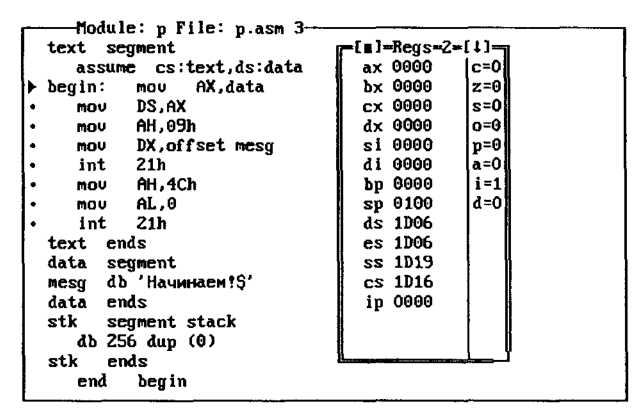


Рис. 5.3. Сегментные регистры после загрузки программы в память

Теперь выполним два первых предложения программы, в которых в регистр DS засылается сегментный адрес сегмента данных. Мы увидим, что содержимое регистра DS стало равно 1D18h. Таким образом, сегменты программы расположены в памяти в том же порядке, что и в исходном тексте: сегмент команд (сегментный адрес 1D16h), сегмент данных (адрес 1D18h), сегмент стека (адрес 1D19h). Однако сразу после загрузки программы в память содержимое регистров DS и ES составляет 1D06h, т. е. на 10h меньше, чем сегментный адрес сегмента команд. Это значит, что перед сегментом команд расположен еще один сегмент, имеющий размер 10h параграфов или 100h=256 байт. Это уже упоминавшийся ранее префикс программы PSP.

Заглянув теперь в листинг трансляции, мы увидим, что размер сегмента команд составляет 12h=18 байт, а размер сегмента данных – 0Ah=10 байт. Имея все эти данные, легко построить чертеж, отображающий расположение в памяти нашей конкретной программы (рис. 5.4).

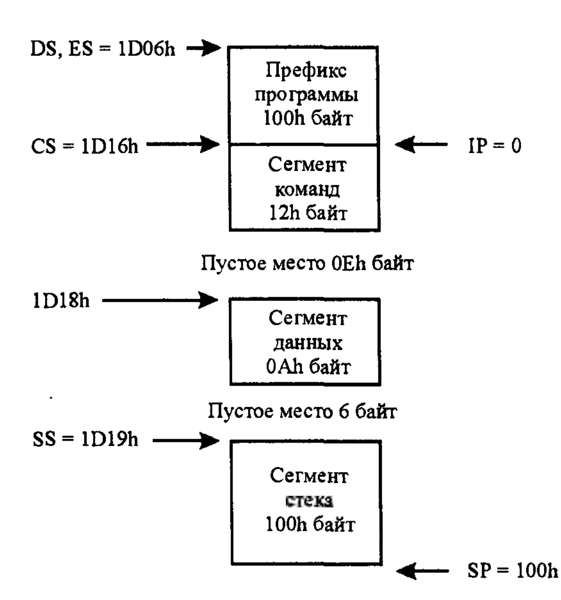


Рис. 5.4. Образ конкретной программы в памяти и исходное содержимое регистров

Перед сегментом команд и вплотную к нему система размещает дополнительный системный сегмент – префикс программы PSP. При загрузке программы в память оба сегментных регистра данных DS и ES указывают на PSP, что дает возможность обращаться к этому сегменту по ходу программы, если это будет нужно. Сегментный регистр CS указывает на сегмент команд, как это и должно быть. Поскольку наша программа начинает свое выполнение с первого предложения (выше отмечалось, что так бывает не всегда), указатель команд IP получает значение 0. В качестве начального значения IP система принимает смещение программной метки, указанной в последнем предложении программы

```
end     begin
```

В нашем случае значение begin равно нулю, так как ею помечена первая команда сегмента команд, имеющая смещение 0 (см. рис. 4.6).

В сегментный регистр SS система загружает сегментный адрес стека. Одновременно указатель стека SP получает значение, соответствующее размеру стека. Автоматическая настройка регистров SS:SP произошла потому, что при описании сегмента стека в программе мы использовали ключевое слово stack:

```
stk     segment stack
```

Все сегменты программы позиционируются на границы параграфов, т. е. начинаются с физических адресов, кратных 16. Поскольку размеры сегментов обычно не кратны 16, между ними в памяти возникают промежутки, неиспользуемые при выполнении программы. Один из таких промежутков можно увидеть на рис. 4.6, где вслед за последней командой программы int 21h, расположенной по адресу CS:0010, начинается область, заполненная нулями. Отладчик, не зная, что эти нули представляют собой просто "мусор", попытался деассемблировать их в команду add [BX+SI],AL, код которой действительно равен нулю.

Если бы мы расположили сегменты в другом порядке, например начали бы программу с сегмента данных (что часто делают, так как для программиста естественно

сначала описать все данные, используемые в программе, а затем уже действия над ними), то сегменты загрузились бы в память в таком порядке:

PSP – Сегмент данных – Сегмент команд – Сегмент стека

Таким образом, префикс программы всегда примыкает к ее самому первому сегменту.

Чрезвычайно важным обстоятельством является то, что после загрузки программы в память оба сегментных регистра данных, и DS и ES, указывают на PSP, а сегмент данных программы оказывается неадресуемым. Не забывайте об этом! Если вы позабудете инициализировать регистр DS так, как это сделано в предложениях 3 и 4 примера 1.1, вы не сможете обращаться к своим данным. При этом транслятор не выдаст никаких ошибок, но программа будет выполняться неправильно. Поставьте поучительный эксперимент: уберите из текста программы 1.1 строки инициализации регистра DS (проще всего не стирать эти строки, а поставить в их начале знак комментария – символ точки с запятой). Оттранслируйте, скомпонуйте и выполните такой вариант программы. Ничего ужасного не произойдет, но на экран будет выведена какая-то ерунда. Возможно, в конце этой ерунды будет и строка "Начинаем!". Почему так получилось? Когда начинает выполняться функция DOS 09h, она предполагает, что полный двухсловный адрес выводимой на экран строки находится в регистрах DS:DX (в DS – сегментный адрес, в DX – смещение). У нас же сегментный регистр DS указывает на PSP. В результате на экран будет выводиться содержимое PSP, который заполнен адресами, кодами команд и другой числовой (а не символьной) информацией.

Для закрепления изложенного материала рассмотрим пример несколько нетипичной программы, назначение которой – вывод на экран даты выпуска ПЗУ BIOS, установленной на данном компьютере (подробнее о ПЗУ BIOS рассказано в статье 22). Известно, что эта дата записана в ПЗУ BIOS в символьной форме (т. е. в виде текста) в 8 байтах, первый из которых имеет смещение FFF5h от начала ПЗУ. Микросхема ПЗУ BIOS располагается в самом конце адресного пространства МП-86 и всегда имеет сегментный адрес F000h. Для вывода содержимого ПЗУ на экран нет необходимости переписывать ячейки ПЗУ в буфер программы; достаточно указать для вызываемой функции DOS в качестве адреса исходной строки значение F000h:FFF5h. Как уже отмечалось выше, функции DOS вывода на экран требуют, чтобы адрес выводимой строки находился в регистрах DS:DX (еще раз подчеркнем, что адрес памяти всегда состоит из двух слов и, следовательно, требует для записи двух регистров, первый из которых должен быть сегментным). Таким образом, перед вызовом DOS следует записать в регистр DS число F000h, а в регистр DX – FFF5h.

В предыдущем примере для вывода на экран использовалась функция DOS 09h. Однако она требует, чтобы в конце выводимой строки стоял знак $. В ПЗУ BIOS такого знака, естественно, нет. Поэтому для вывода на экран нам придется воспользоваться другой функцией DOS, с номером 40h, которая позволяет задавать число выводимых байтов в регистре CX. Это универсальная функция, с помощью которой можно вывести данные на любое устройство (экран, принтер, файл на диске и последовательный порт). Приемное устройство характеризуется значением дескриптора, которое записывается в регистр BX. Для экрана предусмотрен предопределенный дескриптор 1 (последовательному порту присвоен дескриптор 3, принтеру – 4, а бóльшие номера назначаются открываемым файлам).

Пример 5.1. Программа, выводящая на экран дату выпуска ПЗУ BIOS

```
text    segment              ;(1)Начало сегмента команд
        assume CS:text       ;(2)CS будет указывать на сегмент text
begin:  mov    AX,0F000h     ;(3)Сегментный адрес ПЗУ BIOS
        mov    DS,AX         ;(4)Занесем его в DS
        mov    AH,40h        ;(5)Функция DOS вывода
        mov    BX,1          ;(6)Дескриптор экрана
        mov    CX,8          ;(8)Выведем 8 символов
        mov    DX,0FFF5h     ;(7)Смещение к выводимой строке
        int    21h           ;(9)Вызов DOS
        mov    AX,4C00h      ;(10)Функция 4Ch и код завершения 0
        int    21h           ;(11)Вызов DOS
text    ends                 ;(12)Конец сегмента команд
stk     segment              ;(13)Начало сегмента стека
        db 256 dup (0)       ;(14)Стек
stk     ends                 ;(15)Конец сегмента стека
        end    begin         ;(16)Конец текста с точкой входа
```

Приведенная программа своеобразна в том отношении, что в ней отсутствует сегмент данных. Это вполне допустимо. Действительно, если в программе нет данных (они у нас хранятся в ПЗУ), то зачем ей сегмент данных? Поэтому и в предложении assume оставлено только описание регистра CS.

В первых предложениях программы (3 и 4) как и раньше, выполняется инициализация сегментного регистра DS, однако он настраивается не на сегмент данных, а на сегмент памяти, в котором размещается ПЗУ BIOS. Далее настраиваются регистры BX, CX и DX, с помощью которых DOS передается необходимая для правильного выполнения функции информация, после чего командой int 21h вызывается DOS.

Как уже подчеркивалось выше, любая программа обязательно должна завершаться вызовом функции 4Ch. Эта функция обеспечивает правильное завершение программы и возврат в DOS (конкретно – в командный процессор). Если позабыть вызвать в конце программы эту функцию, процессор, выполнив последнее (с нашей точки зрения) предложение программы, продолжит выполнение тех кодов, которые находятся в памяти вслед за сегментом данных. В нашем случае там находится стек, который может быть заполнен чем угодно. Выполнение этих бессмысленных кодов рано или поздно приведет к "зависанию" программы и необходимости перезагрузки компьютера.

Статья 6. **Стек**

В предыдущих статьях вскользь упоминался стек. Рассмотрим это понятие более подробно.

Стеком называют область программы для временного хранения произвольных данных. Отличительной особенностью стека является своеобразный порядок выборки содержащихся в нем данных: в любой момент времени в стеке доступен только верхний элемент, т. е. элемент, загруженный в стек последним. Выгрузка из стека верхнего элемента делает доступным следующий элемент.

Элементы стека располагаются в области памяти, отведенной под стек, начиная со дна стека (т. е. с его максимального адреса), по последовательно уменьшающимся адресам (рис. 6.1). Адрес верхнего, доступного элемента хранится в регистре-указателе стека SP. Как и любая другая область памяти программы, стек должен входить в какой-то сегмент или образовывать отдельный сегмент. В любом случае сегментный адрес этого сегмента помещается в сегментный регистр стека SS. Таким образом, пара

регистров SS:SP описывают адрес доступной ячейки стека: в SS хранится сегментный адрес стека, а в SP – относительный адрес доступной (текущей) ячейки (рис. 6.1, *а*). Обратите внимание на то, что в исходном состоянии указатель стека SP указывает на ячейку, лежащую под дном стека и не входящую в него (на рис. 6.1 эта ячейка обозначена серым цветом).

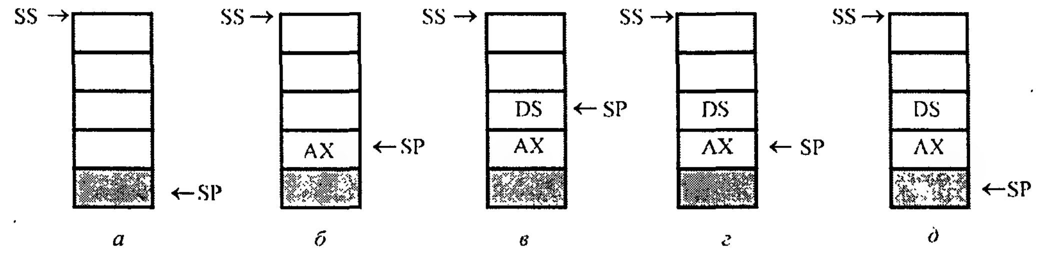


Рис. 6.1. Организация стека: а – исходное состояние; б – после загрузки первого элемента (в данном примере – содержимого регистра AX); в – после загрузки второго элемента (содержимого регистра DS); г – после выгрузки одного элемента; д – после выгрузки двух элементов и возврата в исходное состояние

Загрузка в стек осуществляется специальной командой работы со стеком push (протолкнуть). Эта команда сначала уменьшает на 2 содержимое указателя стека, а затем помещает операнд по адресу, находящемуся в SP. Если, например, мы хотим временно сохранить в стеке содержимое регистра AX, следует выполнить команду

```
push   AX
```

Стек переходит в состояние, показанное на рис. 6.1, *б*. Видно, что указатель стека смещается на 2 байта вверх и по этому адресу записывается указанный в команде проталкивания операнд. Следующая команда загрузки в стек, например

```
push   DS
```

переведет стек в состояние, показанное на рис. 6.1, *в*. В стеке будут теперь храниться два элемента, причем доступным будет только верхний, на который указывает указатель стека SP. Если спустя какое-то время нам понадобилось восстановить исходное содержимое сохраненных в стеке регистров, мы должны выполнить команды выгрузки из стека pop (вытолкнуть):

```
pop    DS
pop    AX
```

Состояние стека после выполнения первой команды показано на рис. 6.1, *г*, а после второй – на рис. 6.1, *д*. Для правильного восстановления содержимого регистров выгрузка из стека должна выполняться в порядке, строго противоположном загрузке – сначала выгружается элемент, загруженный последним, затем предыдущий элемент и т. д.

Обратите внимание на то, что после выгрузки сохраненных в стеке данных они физически не стерлись, а остались в области стека на своих местах. Правда, при "стандартной" работе со стеком они оказываются недоступными. Действительно, поскольку указатель стека SP указывает под дно стека, стек считается пустым; очередная команда push поместит новое данное на место сохраненного ранее содержимого AX, затерев его. Однако пока стек физически не затерт, сохраненными и уже выбранными из него данными можно пользоваться, если помнить, в каком порядке они расположены в сте-

ке. Этот прием часто используется при работе с подпрограммами и в дальнейшем будет описан подробнее.

В примерах 1.1 и 5.1 мы не использовали команды push и pop для работы со стеком. Однако это совсем не означает, что мы могли отказаться от стека. Действительно, стек автоматически используется системой в ряде случаев, в частности при переходе на подпрограммы и при выполнении команд прерывания int. И в том и в другом случае процессор заносит в стек адрес возврата, чтобы после завершения выполнения подпрограммы или программы обработки прерывания можно было вернуться в ту точку вызывающей программы, откуда произошел переход. Подпрограмм в наших программах не было, однако были две команды int 21h и операционная система при выполнении программы дважды обращалась к стеку.

Какой размер должен быть у стека? Точно ответить на этот вопрос нельзя. Если изучить программы DOS, реализующие функции общего назначения, то можно определить, что они перед переходом на выполнение затребованной функции сохраняют в стеке, помимо адреса возврата, еще и содержимое всех регистров процессора. Для этого требуется 15 слов, и, следовательно, стек никак не должен быть меньшего размера. Если же в программе стек используется в явном виде, например для передачи в вызываемые подпрограммы их параметров (которых может быть и очень много), то программист должен сам оценить, какой объем данных может оказаться в стеке в самом неблагоприятном случае, и выбрать размер стека, исходя из этой величины. В наших простых примерах мы отводили под стек 256 байт (128 слов), что, безусловно, достаточно.

Что произойдет, если программист по ошибке или умышленно не опишет стек в своей программе? Модифицируйте пример 1.1, убрав из него предложения описания стека. Запустите получившуюся программу под управлением отладчика. Посмотрите, чему равно содержимое регистров SS и SP. Вы увидите, что в SS находится тот же адрес памяти, что и в CS; отсюда можно сделать вывод, что сегменты команд и стека совпадают. Однако содержимое SP равно нулю. Первая же команда PUSH уменьшит содержимое SP на 2, т. е. поместит в SP число –2. Значит ли это, что стек будет расти, как ему и положено, вверх, но не внутри сегмента команд, а над ним, по адресам –2, –4, –6 и т. д. относительно верхней границы сегмента команд? Оказывается, это не так.

Если взять 16-разрядный двоичный счетчик, в котором записан 0, и послать в него два вычитающих импульса, то после первого импульса в нем окажется число FFFFh, а после второго – FFFEh. При желании мы можем рассматривать число FFFEh как –2 (что и имеет место при работе со знаковыми числами, о которых будет идти речь позже), однако процессор при вычислении адресов рассматривает содержимое регистров как целые числа без знака и число FFFEh оказывается эквивалентным не –2, а 65 534. В результате первая же команда занесения данного в стек поместит это данное не над сегментом команд, а в самый его конец, в последнее слово по адресу CS:FFFEh. При дальнейшем использовании стека его указатель будет смещаться в сторону меньших адресов, проходя значения FFFCh, FFFAh и т. д.

Таким образом, если в программе отсутствует явное объявление стека, система сама создает стек по умолчанию в конце сегмента команд или, точнее, по адресу FFFEh относительно начала сегмента команд (рис. 6.2).

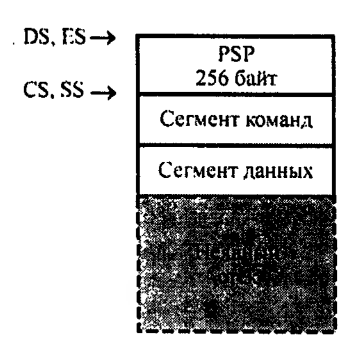


Рис. 6.2. Стек в программе без явного объявления стека

Рассмотренное явление, когда при уменьшении адреса после адреса 0 у нас получился адрес FFFFh, т. е. от начала сегмента мы прыгнули сразу в его конец, носит название циклического возврата или оборачивания адреса. С этим явлением приходится сталкиваться довольно часто.

Расположение стека в конце сегмента команд не приводит к каким-либо неприятностям, пока размер программы далек от граничной величины 64 К. В этом случае начало сегмента команд занимают коды команд, конец – стек, а между ними располагаются данные (если сегмент данных описан в программе после сегмента команд). Если, однако, размер сегментов команд или данных приближается к 64 К, то фактически стек будет наложен на тот или иной сегмент программы и возникает опасность затирания полей программы. Очевидно, что этого нельзя допускать. В то же время система не проверяет, что происходит со стеком и никак не реагирует на затирание команд или данных. Таким образом, оценка размеров собственно программы, данных и стека является важным этапом разработки программы.

Примеры использования стека будут приведены в последующих статьях.

Статья 7. **Вызовы DOS и их использование в прикладных программах**

На каком бы языке ни была написана программа, вся ее жизнь проходит в тесном взаимодействии с установленной на компьютере операционной системой. Даже если программа, в сущности, ничего не делает (нетрудно написать такую "фиктивную" программу), то ее загрузку в память, запуск и завершение выполняют программы DOS. Выше мы уже отмечали, что необходимым элементом любой программы должен быть вызов функции DOS 4Ch, которая завершает программу и передает управление командному процессору. Помимо запуска и завершения программ, в прерогативу DOS входит организация взаимодействия со всеми аппаратными средствами компьютера: клавиатурой, экраном, дисками, таймером, памятью и др. Более того, сама внутренняя организация программы, ее структура и в известной степени алгоритмы поведения в сильной степени определяются правилами организации вычислительного процесса, заложенными в DOS. Так, например, средствами DOS создаются резидентные программы, без которых невозможно себе представить функционирование компьютера. Выше уже приводился пример обращения к окружению программы, которое, разумеется, организует DOS. Еще один пример – запуск программы с передачей ей в командной строке каких-либо параметров, например имен рабочих файлов. Взаимодействие про-

граммы с командной строкой ее запуска также организует DOS. Поэтому в программах на языке ассемблера всегда широко используются системные средства (функции DOS).

При составлении программы на языке высокого уровня (Паскаль, Си) к явным вызовам функций DOS прибегают редко. Однако функции языка, так или иначе затрагивающие аппаратуру, например функции ввода с клавиатуры, вывода на экран, создания файла и др., в процессе компиляции преобразуются в те же вызовы функций операционной системы. Знакомство с внутренними возможностями DOS позволяет более осознанно подойти к составлению программ на используемом языке высокого уровня. Если же программа составляется на языке ассемблера, то без обращения к функциям DOS ее написать просто невозможно.

Мы уже знаем, что вызов DOS осуществляется с помощью команды прерывания int 21h. Как выполняется команда int и почему ее выполнение приводит к активизации программ операционной системы, будет рассмотрено в статье 25, посвященной системе прерываний компьютера. Пока отметим только, что эта команда передает управление программе-диспетчеру DOS, который, проанализировав содержимое регистра AH, вызывает программу, реализующую ту функцию DOS, номер которой совпадает с содержимым AH. Программист, организуя в своих программах вызовы функций DOS, должен иметь перед собой справочник с номерами и описаниями этих функций.

Обращение из прикладной программы к системным функциям осуществляется единообразно. В регистр AH засылается номер функции, в другие регистры – исходные данные, необходимые для выполнения конкретной системной программы. После этого выполняется команда int21h.

Довольно часто возникает ситуация, когда запрошенная системная функция не может быть выполнена должным образом. Например, программа пытается открыть файл на диске, которого в действительности на этом диске нет. Или программе требуется дополнительно 200 Кбайт памяти, а свободной памяти осталось меньше. В такого рода ситуациях большинство функций DOS оповещают вызывающую программу о системном сбое (системной ошибке). Это оповещение осуществляется путем установки флага переноса CF в регистре флагов. Если же функция выполнилась успешно, флаг CF сбрасывается. Таким образом, программа после обращения к системе обязана проанализировать состояние флага CF. Если CF=0, запрошенная функция выполнена успешно и программа может продолжаться. Если же CF=1, функция выполнена не была и продолжать программу нет никакого смысла. В последнем случае в регистре AX система возвращает в качестве дополнительной информации код ошибки, по которому можно судить о характере системного сбоя. Методика анализа флага переноса после обращения к системным функциям будет продемонстрирована в последующих примерах.

Рассмотрим пару простых программ, демонстрирующих методику использования функций DOS и анализа результата их выполнения.

Пример 7.1. Удаление файла

```
text    segment
assume cs:text,ds:data
begin:  mov    AX,data          ;Инициализируем
        mov    DS,AX            ;регистр DS
        mov    AH,41h           ;Функция удаления файла
        mov    DX,offset fname  ;Адрес имени файла
        int    21h              ;Вызов DOS
        jc     error            ;Переход, если ошибка
        mov    AH,09h           ;Вывод сообщения
```

```
         mov    DX,offset msgok    ;об удалении
         int    21h                ;Вызов DOS
fin:     mov    AX,4C00h           ;Завершение программы
         int    21h                ;Вызов DOS
error:   mov    AH,09h             ;Вывод сообщения
         mov    DX,offset msgerr   ;об ошибке
         int    21h                ;Вызов DOS
         jmp    fin                ;Переход на завершение
text     ends
data     segment
fname    db 'f:\testfile.001',0    ;Спецификация файла
msgok    db 'Файл удален$'
msgerr   db 'Файл не найден$'
data     ends
stk      segment stack
         db     256 dup (0)
stk      ends
         end    begin
```

Пример 7.1 представляет собой типичную программу, содержащую три сегмента: команд, данных и стека. В ней демонстрируется методика программного удаления с диска файла с заданным именем. Для удаления файлов предусмотрена функция DOS 41h. Она требует единственного параметра – полной спецификации удаляемого файла, включающей диск, полный путь к файлу и его имя с расширением. Адрес спецификации файла должен быть занесен в регистры DS:DX. Если по каким-либо причинам DOS не смогла удалить файл (файл отсутствует, в спецификации файла указан несуществующий каталог, файл защищен от удаления), функция возвращает установленный флаг переноса CF.

Полная спецификация файла описывается в сегменте данных в строке с произвольным именем fname. Обратите внимание на завершающий строку двоичный нуль, который нужен для того, чтобы DOS могла найти конец этой строки. Такой формат символьных строк используется очень часто и носит специальное название строк ASCIIZ (в отличие от строк ASCII, в которых нуля в конце нет). В описании функции 41h специально указано, что спецификация файла должна быть дана в виде строки ASCIIZ.

Далее в сегменте данных описаны две строки с сообщениями о нормальном завершении (файл удален) и об ошибке (DOS не смогла удалить файл).

Сама программа весьма проста. В первых двух предложениях регистр DS инициализируется адресом сегмента данных. Это нужно для правильной работы как функции 09h, которая требует адрес выводимой на экран строки в регистрах DS:DX, так и функции 41h, которая в той же паре регистров ожидает найти адрес спецификации удаляемого файла.

Далее задаются параметры для выполнения функции удаления файла (в регистр AH заносится номер функции, а в регистр DX – смещение строки со спецификацией файла) и командой int 21h осуществляется переход в DOS. Командой jc (jump if carry, переход, если перенос) осуществляется переход на метку error в случае ошибки. Если ошибки нет, выполняются следующие программные строки – вывод с помощью уже знакомой нам функции 09h сообщения об удалении файла и завершение программы вызовом функции 4Ch с кодом завершения 0.

Если DOS не смогла удалить указанный файл, происходит переход на метку error. Здесь выводится сообщение об ошибке и командой jmp (jump, переход) осуществляется переход на метку fin, которой помечен фрагмент программы с вызовом функ-

ции 4Ch. Последний прием позволяет избавиться от дублирования фрагмента завершения в конце текста программы.

В приведенном примере имеется характерная деталь: программа, если можно так выразиться, кончается не в конце. Действительно, предложения завершения программы

```
fin:    mov   AX,4C00h     ;Завершение программы
        int   21h          ;Вызов DOS
```

расположены внутри текста программы и не являются завершающими. Это совершенно нормальная и даже весьма обычная ситуация. Надо только следить за тем, чтобы при любом ходе программы управление рано или поздно было передано на эти строки. В нашем случае передача управления осуществляется командой

```
        jmp   fin
```

в последнем (по тексту) предложении программы.

Для того чтобы испытать в действии пример 7.1, следует создать (например, средствами программы Norton Commander) в корневом каталоге диска F: файл с именем TESTFILE.001. Если на компьютере читателя диска F: нет, придется подправить текст программы и в спецификации файла указать любой имеющийся на компьютере логический диск, например C: (или, при желании, A:, если пользователь предпочитает проводить эксперименты с дискетой).

После нормального завершения программы и получения на экране сообщения

Файл удален

следует запустить программу вторично. Поскольку при этом прогоне DOS не обнаружит указанный в программе файл, на экран будет выдано сообщение об ошибке:

Файл не найден

Приведенный пример можно усовершенствовать, введя в него анализ кода ошибки в случае возврата DOS установленного флага CF. Функция 41h может вернуть в регистре AX следующие коды ошибки:

02h – файл не найден;

03h – путь не найден;

05h – доступ запрещен.

В приводимом ниже примере 7.2 для каждой из возможных ошибочных ситуаций на экран выводится соответствующее сообщение.

Пример 7.2. Удаление файла с анализом возможных системных сбоев

```
text    segment
assume  cs:text,ds:data
begin:  mov   AX,data
        mov   DS,AX
        mov   AH,41h
        mov   DX,offset fname
        int   21h
        jc    error           ;Переход, если ошибка
        mov   AH,09h          ;Вывод сообщения
        mov   DX,offset msgok ;об удалении
        int   21h
fin:    mov   AX,4C00h        ;Завершение программы
        int   21h
;Блок анализа ошибок
error:  cmp   AX,02h          ;Файл не найден?
        je    notfound        ;Да, вывести сообщение
        cmp   AX,03h          ;Путь не найден?
```

```
        je     wrongdir          ;Да, вывести сообщение
        cmp    AX,05h            ;Доступ запрещен?
        je     noaccess          ;Да, вывести сообщение
        jmp    fin               ;Неизвестная ошибка
;Блок вывода сообщений
notfound:mov   DX,offset msg1
        jmp    write
wrongdir:mov   DX,offset msg2
        jmp    write
noaccess:mov   DX,offset msg3
write:  mov    AH,09h
        int    21h
        jmp    fin
text    ends
data    segment
fname   db 'f:\testfile.001',0
msgok   db 'Файл удален$'
msg1    db 'Файл не найден$'
msg2    db 'Каталог неверен$'
msg3    db 'Доступ запрещен$'
data    ends
stk     segment stack
        db     256 dup (0)
stk     ends
        end    begin
```

В этом варианте программы в случае установки флага переноса и перехода на метку error выполняется сравнение содержимого регистра AX с возможными для данной функции кодами ошибки. Сравнение осуществляется командой cmp (compare, сравнить), после которой командой условного перехода je (jump if equal, переход, если равно) выполняется переход на соответствующую этой ошибке метку блока вывода сообщений. Если ни одно из сравнений не дало положительного результата, т. е. функция 41h вернула непредусмотренный код ошибки, происходит переход на метку fin и завершение программы без каких-либо сообщений. В данном случае такой поворот событий трудно себе представить, в принципе же в блоках сравнения следует предусматривать отработку всех возможных ситуаций.

В блоке вывода сообщений в регистр DX заносится смещение строки, соответствующей ошибке, и командой jmp осуществляется переход на метку write, где вызывается функция 09h вывода строки на экран.

Для полной проверки правильности созданной программы следует прогнать ее в разных ситуациях: файл имеется, файла нет, файл защищен атрибутом "только для чтения" (этот атрибут легко установить с помощью команды DOS ATTRIB или средствами программы Norton Commander), в программе в спецификации файла указан несуществующий диск или каталог. В последнем случае, разумеется, внеся в программу "ошибку", необходимо повторно выполнить ее трансляцию и компоновку.

Мы видим, что анализ даже немногих кодов ошибок привел к существенному увеличению размеров программы и введению в нее многочисленных условных и безусловных переходов. Тем не менее при разработке реальных программных продуктов такого рода проверки абсолютно необходимы. Ведь ошибки при выполнении функций DOS возникают не потому, что программа написана неверно, а потому, что конкретный прогон программы не отвечает состоянию вычислительной среды. Например, программа должна в начале своей работы удалить с дискеты файл, оставшийся там от предыдущего сеанса работы, а пользователь случайно поставил на дисковод не ту дис-

кету. Достаточно установить правильную дискету – и программа будет работать нормально. Если в программе не выполняется анализ ошибок DOS, то программа в таких ошибочных ситуациях может начать работать совершенно непредсказуемо.

Представьте себе программу, которая считывает с диска некоторые данные, обрабатывает их и создает новый файл с этими обработанными данными. Что будет, если исходного файла на диске не окажется? Файл открыт не будет, и DOS сообщит нам об ошибке установкой флага переноса. Если мы не предусмотрим в программе анализа флага ошибки, то программа просто продолжит свое выполнение. Будет предпринята попытка чтения из отсутствующего файла, но файла нет, функция чтения выполнена не будет, и опять DOS вернет установленный флаг CF. Если и здесь мы не обращаем на него внимания, выполнение программы продолжится, она начнет обрабатывать отсутствующие данные и, возможно, запишет результат своей бесполезной работы в файл на диске. Хотя все выполнение программы не имело никакого смысла, мы ничего об этом не узнаем и будем думать, что обладаем файлом с ценными данными.

Некоторые простые функции DOS не сигнализируют о возможных ошибках установкой флага CF. Например, функция 01h вводит один символ с клавиатуры, а функция 02h выводит один символ на экран. Трудно представить себе ситуацию, в которой эти функции (на нормально работающем компьютере) дали бы сбой. Таким образом, используя те или иные функции DOS, следует по справочнику определить, возвращают ли они информацию о сбое, и если возвращают, то с какими кодами в регистре AX.

Статья 8. **Циклы**

Циклы (т. е. выполнение некоторого участка программы заданное число раз) относятся к числу важнейших элементов программ на любых языках программирования. Для демонстрации техники организации циклов рассмотрим фрагмент программы, в котором создается и выводится на экран тестовый символьный массив, заполненный кодами алфавитно-цифровых и псевдографических символов. Эти символы имеют коды от 32 (пробел) до 254 (сплошной квадратик). (Примем пока это утверждение на веру; вопрос о кодах символов будет подробнее рассмотрен в статье 11). Такой массив можно создать в полях данных программы вручную с помощью оператора db:

```
symbols db    32,33,34,35,36,37,38,39,...
```

однако проще заполнить его данными программным образом (пример 8.1). Для удобства читателей программа примера 8.1 приведена полностью, однако в дальнейшем в примерах будут приводиться только содержательные фрагменты программ. Весь "антураж" (объявления сегментов, инициализация сегментного регистра DS, завершение программы), который всегда остается практически неизменным, будет опускаться.

Пример 8.1. Циклы

```
text     segment              ;(1)Начало сегмента команд
assume   CS:text,DS:data      ;(2)
begin:   mov    AX,data       ;(3)Инициализация сегментного
         mov    DS,AX         ;(4)регистра DS
;Подготовим все необходимое для организации цикла
         mov    CX,223        ;(5)Число шагов в цикле
         mov    SI,0          ;(6)Индекс элемента в заполняемом массиве
         mov    AL,32         ;(7)Код первого символа - пробела
;Теперь собственно цикл, в который входит 4 команды
```

```
fill:    mov    symbols[SI],AL;(8)Очередной код в байт массива
         inc    AL            ;(9)Создадим код следующего символа
         inc    SI            ;(10)Сдвинемся в массиве на 1 байт
         loop   fill          ;(11)Команда цикла из CX шагов
;Выведем для контроля полученный символьный массив на экран
         mov    AH,40h        ;(12)Функция DOS вывода
         mov    BX,1          ;(13)Стандартный дескриптор экрана
         mov    CX,223        ;(14)Число выводимых байтов
         mov    DX,offset symbols;(15)Адрес выводимого сообщения
         int    21h           ;(16)Вызов DOS
;Завершим программу
         mov    AX,4C00h      ;(17)
         int    21h           ;(18)
text     ends                 ;(19)Конец сегмента команд
data     segment              ;(20)Начало сегмента данных
;Поля данных программы
symbols db 223 dup ('*')      ;(21)Заполняемый массив
data     ends                 ;(22)Конец сегмента данных
stk      segment stack        ;(23)Начало сегмента стека
         db     256 dup (0)   ;(24)Стек
stk      ends                 ;(25)Конец сегмента стека
         end    begin         ;(26)Конец текста программы
```

Рассмотрим содержательную часть программы. Счетчиком шагов цикла служит регистр CX, в который надо занести требуемое число шагов цикла (предложение 5), равное длине заполняемого массива. Работа с массивом осуществляется, как правило, с помощью одного из индексных регистров (SI или DI), в которых хранится и наращивается индекс адресуемого элемента массива, т. е. номер байта массива, к которому осуществляется обращение в данном шаге цикла. Поскольку мы начинаем обрабатывать массив с самого начала, в предложении 6 в регистр SI заносится 0. Регистр AL выбран нами для хранения текущего кода символа, отправляемого в массив. С таким же успехом эту роль мог бы выполнить любой другой байтовый регистр – AH, BL, BH, DL или DH (регистры CL и CH, входящие в состав регистра CX, уже заняты). В предложении 7 в регистр AL заносится код первого символа – пробела. Подготовив все необходимые регистры, можно составить само тело цикла. В предложении 8 код из AL отправляется в элемент (байт) массива symbols, номер которого определяется содержимым индексного регистра SI. Это так называемая индексная адресация, у которой существует несколько разновидностей. В частности, в качестве индексного регистра с тем же успехом можно было использовать BX или DI. В первом шаге цикла заполнится элемент массива с индексом 0. В следующих двух предложениях командой inc (increment, инкремент) выполняется увеличение на 1 кода очередного символа и индекса в массиве. Наконец, команда loop (петля) (предложение 11) возвращает управление на метку fill, причем делает это ровно 223 раза, в соответствии с исходным содержимым CX. Заметим попутно, что в качестве метки может использоваться любое обозначение, начинающееся с буквы.

Чтобы увидеть результаты работы программы, выведем полученный массив на экран. Мы уже знаем, что вывод на экран осуществляется с помощью функций DOS, причем в предыдущих примерах использовались две функции – 40h (пример 5.1) и 09h (все остальные). Функцию 09h использовать здесь нельзя, потому что где-то в массиве будет содержаться код знака $, а он не может быть выведен на экран функцией 09h. Поэтому воспользуемся функцией 40h, для которой необходимо занести в регистр BX дескриптор экрана (код 1), в CX – число выводимых байтов, а в DX – адрес выводимой

информации (предложения 13...15). Команда int 21h передает управление DOS, которая и выполняет требуемую операцию.

В полях данных программы объявлен массив байтов (оператор db), который инициализирован кодами символа *. Это очень удобный и распространенный прием, который облегчает отладку программы. Если в силу каких-то ошибок в программе массив будет заполнен не весь или не заполнен вообще, на экран будут выведены звездочки. Если же программа работает правильно, исходные звездочки затрутся заполняющими символами.

В заключение отметим еще одну особенность программы 8.1. Это первая программа, в которой выполняется прямое обращение к ячейкам сегмента данных (предложение 8, где заполняется массив с именем symbols). В предыдущих статьях подчеркивалось, что адрес любой ячейки памяти обязательно имеет два компонента: сегментный адрес, хранящийся в одном из сегментных регистров, и смещение, которое, в частности, может указываться в команде в виде мнемонического обозначения ячейки. В предложении 8 имеется ссылка на ячейку symbols, т. е. указывается смещение. Каким сегментным регистром будет пользоваться процессор при выполнении этой команды? Если в команде не указан в явной форме сегментный регистр, по умолчанию используется DS. Собственно, именно в этом предположении мы в начале программы инициализировали регистр DS адресом сегмента данных, в котором расположен массив symbols. Однако для того, чтобы указанное умолчание действовало, необходимо с помощью оператора assume указать соответствие DS именно этому сегменту. Таким образом, определение DS:data в операторе assume стало необходимым только в этой программе, во всех предыдущих можно было обойтись без него.

Рассмотрим теперь вложенные циклы на примере организации программной задержки. Программные задержки используются в тех случаях, когда в какой-то точке программы надо приостановить ее выполнение на некоторое время. Такая необходимость может возникнуть при программировании относительно медленной аппаратуры. Если в аппаратуру посылается последовательно несколько управляющих команд, то для того, чтобы дать аппаратуре время их выполнить, между ними включаются программные задержки (обычно на несколько микросекунд). Задержки значительно большей величины, порядка нескольких секунд, удобно использовать при отладке программ, выводящих на экран некоторую (возможно, отладочную) информацию. Задержка дает возможность программисту проследить результаты выполнения каждого шага программы. В примере 8.2 приведен фрагмент именно такой программы.

Пример 8.2. Программная задержка

```
;Организуем демонстрационный цикл из 10 шагов, которые будут
;выполняться с задержкой порядка нескольких секунд
        mov    CX,10          ;(1)Число шагов в демонстрационном цикле
cycle:  push   CX             ;(2)Сохраним этот счетчик в стеке
;Выведем на экран контрольную строку из трех символов
        mov    AH,09h         ;(3)
        mov    DX,offset string;(4)
        int    21h            ;(5)
;Организуем программную задержку
        mov    CX,100         ;(6)Счетчик внешнего цикла
outer:  push   CX             ;(7)Сохраним его в стеке
        mov    CX,65535       ;(8)Счетчик внутреннего цикла
inner:  loop   inner          ;(9)Повторим команду loop 65535 раз
        pop    CX             ;(10)Восстановим внешний счетчик
        loop   outer          ;(11)Повторим все это 100 раз
```

```
        pop    CX            ;(12)Восстановим счетчик демонстрационного цикла
        loop   cycle         ;(13)Повторим демонстрационный цикл CX=10 раз
;Поля данных (в сегменте данных)
string  db     '<> $'        ;(14)
```

В примере 8.2 с помощью функции DOS 09h (предложения 3...5) на экран выводится строка "<>" с периодом, определяемым программной задержкой. Задержка создается с помощью двух вложенных циклов. Внутренний представляет собой просто команду loop, повторяемую 65 535 раз (предложение 9); внешний цикл служит для повторения внутреннего 100 раз. В результате команда loop выполняется 6 553 500 раз, что в зависимости от скорости конкретного компьютера дает задержку приблизительно от одной до нескольких десятков секунд.

Поскольку команда loop выполняется всегда CX раз, этот регистр приходится использовать в каждом цикле заново. Перед входом во внутренний, вложенный цикл текущее значение счетчика внешнего цикла (содержимое CX) сохраняется в стеке командой push, а перед командой loop внешнего цикла восстанавливается командой pop (пары предложений 2, 12 и 7, 10). Разумеется, сохранить значение CX можно где угодно (в любом другом регистре или в ячейке памяти), однако команды сохранения в стеке и восстановления из стека эффективнее других в смысле времени выполнения и расходуемой памяти.

Подготовив к выполнению пример 8.2, поэкспериментируйте с длительностью задержки, изменяя число шагов внешнего цикла (предложение 6).

Создавать программные задержки с помощью вложенных циклов неудобно. Можно обойтись одним циклом, если вместо регистра CX использовать расширенный регистр ECX. В этом случае максимальное число шагов в цикле составит более 4 млрд. Однако обычное приложение DOS является 16-разрядным, а при выполнении 16-разрядного сегмента команд процессор использует только младшие половины расширенных регистров. Для того чтобы заставить процессор в команде перехода (каковой является команда цикла loop) использовать регистр ECX целиком, следует перед этой командой включить так называемый префикс размера адреса 67h. Тогда фрагмент программной задержки (предложения 6...11 предыдущего примера) существенно упростится:

```
        mov    ECX,600000  ;Счетчик цикла
delay:  db     67h         ;Префикс размера адреса
        loop   delay       ;Повторим команду loop 600 000 раз
```

Однако ассемблер (TASM или MASM) откажется транслировать приведенный фрагмент, сообщив, что ему неизвестно символическое обозначение ECX. Чтобы преодолеть эту трудность, необходимо в начало программы включить директиву .386, а сегменты команд объявить с описателями use16:

```
.386
text    segment use16
...
data    segment use16
```

Директива .386 (можно использовать также директивы .486 или .586) позволит ассемблеру правильно воспринимать обозначения 32-разрядных регистров, а описатели use16 требуют создания обычной 16-разрядной программы. При отсутствии этих описателей ассемблер создаст 32-разрядное приложение, которое в среде MS-DOS работать не будет.

Статья 9. **Прерывания BIOS**

Как известно, операционная система персонального компьютера состоит из двух основных компонентов – базовой системы ввода-вывода BIOS, обеспечивающей управление периферийным оборудованием компьютера, и дисковой операционной системы MS-DOS, в функцию которой входит организация всех элементов вычислительного процесса: запуска и завершения задач, управления памятью, обслуживания файловой системы, службы времени, обработки ошибок и т. д.

Базовая система ввода-вывода, размещаемая в постоянном запоминающем устройстве (ПЗУ BIOS), включает набор управляющих программ для всех основных периферийных устройств компьютера, таких, как магнитные диски, клавиатура, видеосистема, принтер, последовательный порт, часы. Программы BIOS, обеспечивая управление аппаратурой на самом низком, "физическом" уровне, путем обращения к портам, регистрам и аппаратным буферам, являются аппаратно-зависимыми, поэтому микросхемы BIOS разных модификаций машин типа IBM PC могут отличаться друг от друга. Программы DOS, размещаемые в файлах IO.SYS и MSDOS.SYS, образуют более высокий уровень управления компьютером. Так, если для записи данных на диск с помощью программ BIOS требуется задание номеров головки, цилиндра и сектора на конкретном дисководе, то при обращении к DOS достаточно указать спецификацию файла. Программы обслуживания файловой системы, входящие в состав DOS, проанализируют содержимое диска, определят местонахождение требуемого файла и поставят ряд запросов к BIOS на выполнение операций записи. Таким образом, DOS располагается в логическом плане между BIOS и программой пользователя, упрощая программные обращения к аппаратуре. Кроме этого, программы DOS обеспечивают ряд функций, не имеющих прямого отношения к аппаратуре, например динамическое выделение памяти, запуск и завершение программ, обслуживание векторов прерываний и многое другое.

Прикладная программа может с равным успехом обращаться как к функциям DOS, так и к прерываниям BIOS. Использовать функции DOS проще (иногда существенно проще). Однако BIOS обладает большими возможностями. Сравним, например, возможности DOS и BIOS при выполнении таких распространенных операций, как вывод на экран и работа с диском.

Выше приводились примеры вывода текстовых строк на экран с помощью функций DOS 09h и 40h. Эти процедуры весьма просты, однако они позволяют выводить только черно-белый текст; изменить цвет символов или фона под ними нельзя. DOS обеспечивает автоматический переход на следующую строку и прокрутку экрана, однако строки текста приходится выводить сплошным потоком, друг за другом, так как в DOS отсутствуют средства позиционирования курсора. Нельзя, например, вывести некоторую строку в нижнюю часть экрана, а затем заполнить информацией его верхнюю часть. Имеются и другие ограничения: DOS не обеспечивает очистку экрана, переключение видеостраниц, изменение видеорежима. В DOS отсутствуют также средства вывода графических (точечных) изображений.

С другой стороны, BIOS обеспечивает реализацию всех возможностей видеосистемы. Очистка и прокрутка экрана, позиционирование курсора, изменение цвета символов, загрузка шрифтов пользователя, смена видеостраниц и видеорежимов, даже изменение формы курсора – все это доступно с помощью функций прерывания BIOS 10h. Следует также

заметить, что для программ BIOS характерна более высокая скорость работы. Однако за все это приходится платить заметным усложнением программы.

Еще более отчетливо проявляются различия возможностей DOS и BIOS при работе с дисками. DOS предоставляет простые и удобные средства обслуживания файловой системы. При обращении к файлу достаточно указать его имя; DOS возьмет на себя всю работу по поиску на диске всех (часто несвязных) участков, принадлежащих этому файлу, и определению его характеристик, в частности длины. Чтение всего файла или запись в него часто осуществляется единственным вызовом соответствующей функции DOS.

Использование при работе с файлами средств BIOS неоправданно, так как в этом случае на прикладную программу пришлось бы возложить весьма трудоемкую задачу определения физического расположения файлов на диске. С другой стороны, BIOS (конкретно – прерывание 13h) позволяет читать и записывать любые секторы диска. Это дает возможность обращаться к таким областям диска, как загрузочные секторы, таблицы логических дисков, таблицы размещения файлов, каталоги. Правда, к некоторым из этих областей (входящих в пространство логических дисков) можно обратиться и с помощью специально предусмотренных прерываний DOS с номерами 25h и 26h, однако прочитать, например, сектор главного загрузчика или таблицы логических дисков средствами DOS невозможно. Между прочим, некоторые вирусы сохраняют себя в областях диска, выходящих за пределы логических дисков, например на продолжении дорожки 0, вслед за сектором главного загрузчика. Уничтожить такой вирус можно только с помощью прерывания 13h BIOS.

Рассмотрим два примера использования средств BIOS, чтобы подчеркнуть возможности и области использования базовой системы ввода-вывода.

Пример 9.1. Вывод на экран графического изображения с помощью прерывания BIOS 10h

```
;Установим графический видеорежим
        mov     AH,0h       ;(1)Функция установки видеорежима
        mov     AL,10h      ;(2)Графический режим 16 цветов
        int     10h         ;(3)Вызов BIOS
;Выведем на экран желтый прямоугольник
        mov     AH,0Ch      ;(4)Функция вывода пиксела
        mov     AL,0Eh      ;(5)Желтый цвет
        mov     BH,0        ;(6)Видеостраница
        mov     CX,50       ;(7)Начальная x-координата
c2:     mov     DX,10       ;(8)Начальная y-координата
c1:     int     10h         ;(9)Вызов BIOS - вывод точки
        inc     DX          ;(10)Инкремент по y
        cmp     DX,330      ;(11)Дошли до границы по y?
        jne     c1          ;(12)Нет, повторяем вывод точек
        inc     CX          ;(13)Дошли до границы по y, инкремент по x
        cmp     CX,610      ;(14)Дошли до границы по x?
        jne     c2          ;(15)Нет, повторяем вывод вертикальных линий
;Остановим программу для наблюдения результата
        mov     AH,01h      ;(16)Дошли до границы по x, останов
        int     21h         ;(17)функцией ввода символа с клавиатуры
;Переведем систему назад в текстовый режим
        mov     AX,3        ;(18)Восстановим текстовый
        int     10h         ;(19)видеорежим
;Завершим программу
        mov     AX,4C00h    ;(20)
        int     21h         ;(21)
```

В примере 9.1 видеосистема компьютера переводится в графический режим (конкретно – режим EGA с номером 10h – 16 цветов, 350x640 точек) и на экран выводится цветной прямоугольник. Как уже отмечалось, в DOS отсутствуют средства графики, поэтому такую задачу можно решить только с помощью соответствующего прерывания BIOS. Для управления видеосистемой используется прерывание 10h.

Смена видеорежима осуществляется с помощью функции 0; номер видеорежима для этой функции указывается в регистре AL (предложение 2).

Для вывода (по точкам) графического изображения используется функция 0Ch, которая требует целого ряда параметров. В регистр BH заносится номер графической видеостраницы, в регистры CX и DX – x- и y-координаты точки, в регистр AL – ее цвет. Не вдаваясь пока в детали, укажем, что каждому из 16 возможных в режиме EGA цветов соответствует свой код. Так, код 1 обозначает синий цвет, код 2 – зеленый, код 3 – бирюзовый и т. д. Желтому цвету соответствует код 14. Каждый вызов функции 0Ch выводит на экран одну точку. Для вывода прямой линии (горизонтальной или вертикальной) эту функцию следует включить в цикл, а для вывода прямоугольника придется создать два вложенных друг в друга цикла – один по координате x, а другой по координате y. Как уже отмечалось в статье 8, циклы обычно организуются с помощью команды loop, которая неявным образом использует в качестве счетчика цикла регистр CX. Однако у нас регистр CX занят (в нем находится текущая x-координата точки) и цикл придется организовать другим способом. В данном примере внутренний цикл выполняется по координате y, внешний – по координате x, в результате чего прямоугольник выводится вертикальными линиями слева направо.

Командами 4...9 на экран выводится первая точка. В предложении 10 y-координата в регистре DX увеличивается на 1. В предложении 11 выполняется сравнение текущей координаты с (произвольным) конечным значением 330. Если конечное значение еще не достигнуто (DX не равно 330), командой jne (jump if not equal, переход, если не равно) выполняется возврат на метку c1 вызова BIOS и вывод следующей точки (на 1 пиксел ниже предыдущей). В результате во внутреннем цикле (предложения 9...12) выводится вертикальная линия с координатами 10...329. При достижении y-координатой значения 330 выполняется инкремент x-координаты в регистре CX и сравнение ее с конечным значением 610 (предложения 13 и 14). Если конечное значение не достигнуто, командой jne (предложение 15) осуществляется возврат на метку c2, где восстанавливается исходное значение y-координаты и с помощью внутреннего цикла рисуется новая вертикальная линия.

Перед завершением программы следует восстановить исходный текстовый режим. Однако смена режима приведет к очистке видеопамяти и стиранию с экрана выведенного изображения. Поэтому перед восстановлением видеорежима программа останавливается вызовом функции DOS 01h ожидания ввода символа с клавиатуры (предложения 16 и 17). После нажатия на любую клавишу выполняется восстановление текстового режима и завершение программы обычным образом.

Заметьте, что в рассмотренном примере нет данных, хранящихся в памяти и, соответственно, нет необходимости иметь сегмент данных.

В качестве примера использования дискового прерывания BIOS (13h) рассмотрим процедуру чтения и сохранения в файле сектора главного загрузчика жесткого диска (Master boot). Результатом работы этой программы (пример 9.2) можно воспользоваться в случае повреждения главного загрузчика, например вирусом. Запись главного загрузчика из файла на свое место на диске может восстановить поврежденный диск без его переформатирования, которое привело бы к потере всех данных на нем.

Главный загрузчик расположен в самом начале жесткого диска, в секторе 1 цилиндра 0 на поверхности 0. Эта область диска не входит ни в какой логический диск (первый логический диск C: начинается со следующей поверхности с номером 1) и не может быть прочитана средствами DOS. Структура жесткого диска будет подробнее рассмотрена в статье 29.

Пример 9.2. Чтение и сохранение в файле главного загрузчика жесткого диска

```
;Чтение главного загрузчика
        mov     AH,02h          ;(1)Функция чтения секторов
        mov     AL,1            ;(2)Прочитаем 1 сектор
        mov     CH,0            ;(3)Номер цилиндра
        mov     CL,1            ;(4)Номер сектора
        mov     DH,0            ;(5)Номер поверхности
        mov     DL,80h          ;(6)Код жесткого диска
        mov     BX,offset mboot;()Буфер для прочитанного
        int     13h             ;(7)Переход в BIOS
;Создадим файл для сохранения прочитанного
        mov     AH,3Ch          ;(8)Функция (DOS) создания файла
        mov     CX,0            ;(9)Файл будет без атрибутов
        mov     DX,offset fname;(10)Смещение имени файла
        int     21h             ;(11)Переход в DOS
        mov     BX,AX           ;(12)Сохраним в BX дескриптор файла
;Запишем в файл данные из буфера mboot
        mov     AH,40h          ;(13)Функция записи в файл
        mov     CX,512          ;(14)Число выводимых байтов
        mov     DX,offset mboot;(15)Смещение буфера
        int     21h             ;(16)Переход в DOS
;Поля данных (в сегменте данных)
mboot   db      512 dup(0)  ;(17)
fname   db      'mboot.dat',0;(18)
```

Для чтения физических секторов дисков (как жестких, так и гибких) предназначена функция 02h прерывания 13h (предложение 1), которая может читать как один, так и группу следующих друг за другом секторов. Этой функции надо задать число читаемых секторов (у нас 1), полный физический адрес первого из читаемых секторов (т. е. номера цилиндра, сектора и поверхности), код диска (0 для дискеты A, 1 для B и 80h для жесткого диска), а также адрес буфера в программе (mboot, предложение 17), куда будет выполняться копирование содержимого диска.

Выполнив чтение сектора диска в буфер mboot, программа с помощью функции 3Ch DOS (предложение 8) создает на диске новый файл с именем mboot.dat. Этой функции требуется указать атрибуты создаваемого файла (нам атрибуты не нужны) и адрес имени файла. Создав файл, функция 3Ch возвращает в регистре AX дескриптор этого файла, который затем используется при всех обращениях к файлу. Поскольку вызываемая далее в программе функция записи в файл 40h требует, чтобы дескриптор файла содержался в регистре BX, в предложении 12 полученный дескриптор переносится из AX в BX. Наконец, функция 40h выполняет вывод в файл содержимого буфера mboot.

Статья 10. **Способы адресации**

Изучив приведенные в этой книге программы, вы, вероятно, уже обратили внимание на то, что обращение к данным в них выполняется по-разному. Обрабатываемые в программе данные можно помещать непосредственно в регистр, записывать в качестве

одного из операндов прямо в код команды либо тем или иным способом указывать место в памяти, где эти данные расположены. Рассмотрим, например, программу, обеспечивающую поиск максимального элемента в массиве данных.

Пример 10.1. Поиск максимального значения

```
;В сегменте команд
        mov   DL,mas       ;(1)DL=первый элемент массива
        mov   CX,8         ;(2)CX=число элементов минус один (первый)
        mov   BX,1         ;(3)BX=индекс второго элемента
cont:   cmp   DL,mas[BX]   ;(4)DL > следующего элемента?
        jg    next         ;(5)Да, на анализ следующего
        mov   DL,mas[BX]   ;(6)Нет, в DL заносим следующий элемент
next:   inc   BX           ;(7)и на анализ следующего
        loop  cont         ;(8)Цикл до конца массива
;После завершения цикла в DL находится максимальный элемент
        mov   AX,4C00h     ;(9)Вызов функции 4Ch DOS
        int   21h          ;(10)
;В сегменте данных
mas db 1,2,5,30,127,9,8,3,4;(11)Массив с исходными данными
```

Здесь, как и в дальнейших примерах, мы приводим только содержательную часть программы – алгоритм обработки и необходимые для его функционирования поля данных. Подготавливая этот пример к выполнению, оформите его в виде работоспособной программы, описав сегменты команд, данных и стека. Не забудьте также в начале сегмента команд инициализировать регистр DS, а в конце вызвать функцию DOS 4Ch, чтобы после завершения программы передать управление DOS.

Как узнать, работает ли эта программа? До сих пор в наших примерах на экран выводились исключительно символьные (текстовые) строки, заранее описанные в полях данных программы. В настоящем случае результатом работы программы будет последовательность чисел. Просто так вывести число на экран нельзя; для того чтобы получить на экране изображение числа (в двоичном, десятичном или 16-ричном представлении), его необходимо сначала преобразовать в символьную форму. Эта процедура будет рассмотрена в одной из следующих статей; сейчас же для наблюдения результатов работы программы придется обратиться к отладчику.

Запустите программу 10.1 в отладчике и откройте окно регистров (командой Alt+V ➤ R). Уменьшите окно с исходным текстом и расположите оба окна таким образом, чтобы они не заслоняли друг друга. Выполните теперь программу до предложения 9. Проще всего поместить курсор на это предложение и нажать клавишу F4 (выполнение до курсора). Если программа работает правильно, в регистре DL должно оказаться максимальное числа из массива mas, в нашем примере число 127 (=7Fh). Поскольку отладчик не умеет выводить содержимое байтовых регистров, надо посмотреть содержимое регистра DX, которое в данном случае будет равно 007F.

Если программа не работает должным образом, придется выполнить ее в пошаговом режиме. Для этого следует открыть в отладчике окно процессора (Alt+V ➤ C) и, нажимая клавишу F7, выполнять команду за командой, контролируя каждый раз содержимое задействованных в алгоритме регистров процессора (BX, CX и DL).

Обратимся теперь к способам адресации, использованным в примере 10.1.

В предложении 1 в регистр DL заносится содержимое первого байта последовательности данных, которая в этой программе обозначена как mas. Здесь мы сразу же встречаемся с двумя способами адресации данных – регистровым и прямым (прямая адресация к памяти).

При использовании регистровой адресации данные записываются в регистр – 1-байтовый или 2-байтовый.

Прямой способ адресации является представителем группы способов обращения к данным, которые хранятся не в регистрах, а в ячейках памяти. В рассматриваемом случае имя mas фактически является адресом памяти, т. е. смещением к данному в сегменте данных. При этом в качестве сегментного регистра по умолчанию используется регистр DS. Содержимое указанной ячейки памяти переносится (в данном случае) в регистр DL. Если рассматриваемая ячейка памяти располагается в сегменте, базовый адрес которого находится в другом сегментном регистре (ES, CS или SS), то обязательно указание этого регистра:

```
mov DL, ES:mas
```

Еще один способ адресации представлен в предложении 3, где в качестве одного из операндов указано число (конкретно – 1). Такой операнд входит непосредственно в состав команды процессора. Этот способ адресации так и называется – непосредственный.

В предложениях 4 и 6 применена разновидность прямой адресации памяти, в которой указывается не только обозначение ячейки памяти (mas), но и величина сдвига относительно этой ячейки (в данном случае в регистре BX). Очевидно, что такой способ адресации удобен при работе с массивами.

Все имеющиеся способы адресации можно условно разделить на три группы: регистровая, непосредственная и с указанием адреса памяти. При этом адрес памяти можно задавать по-разному: прямым указанием символического обозначения ячейки памяти, указанием регистра, в котором хранится требуемый адрес, или и того и другого. Таким образом, третья группа включает, в сущности, целый ряд способов адресации. Они обычно носят названия: прямая, базовая, индексная, базово-индексная, а также базовая, индексная или базово-индексная со смещением.

Приведем краткий обзор способов адресации.

Регистровая адресация

Операнд (байт или слово) находится в регистре. Способ применим ко всем программно-адресуемым регистрам процессора.

Примеры:

```
        push  DS            ;Сохранение DS в стеке
        mov   BP,SP         ;Пересылка содержимого SP в BP
```

Непосредственная адресация

Операнд (байт или слово) может быть представлен в виде числа, адреса, кода ASCII, а также иметь символьное обозначение.

Примеры:

```
        mov   AX,4C00h      ;Операнд - 16-ричное число
        mov   DX,offset mas;Смещение массива mas заносится в DX
        mov   DL,'!'        ;Операнд - код ASCII символа '!'
num=9                       ;Число 9 получает обозначение num
        mov   CX,num        ;Число, обозначенное num, загружается в CX
```

Прямая адресация памяти

В команде указывается символическое обозначение ячейки памяти, над содержимым которой требуется выполнить операцию.

Пример:

```
mov   DL,mem1     ;Содержимое байта памяти с символическим
                  ;именем mem1 пересылается в DL
```

Если нужно обратиться к ячейке памяти с известным абсолютным адресом, то этот адрес можно непосредственно указать в качестве операнда. Предварительно необходимо настроить какой-либо сегментный регистр на начало того участка памяти, в котором находится искомая ячейка.

Пример:

```
mov   AX,0        ;Настроим сегментный регистр ES на
mov   ES,AX       ;самое начало памяти (адрес 0)
mov   AX,ES:[0]   ;AX=содержимое слова с адресом 0000h:0000h
mov   DX,ES:[2]   ;DX=содержимое слова с адресом 0000h:0002h
```

Заметим, что в этом случае сегментный регистр надо указывать обязательно.

Все остальные способы адресации относятся к группе косвенной адресации памяти.

Базовая и индексная адресация памяти

Относительный адрес ячейки памяти находится в регистре, обозначение которого заключается в квадратные скобки. При использовании регистров BX или BP адресацию называют базовой, при использовании регистров SI или DI – индексной. При адресации через регистры BX, SI или DI в качестве сегментного регистра подразумевается DS; при адресации через BP – регистр SS. Таким образом, косвенная адресация через регистр BP предназначена для работы со стеком. Однако при необходимости можно явно указать требуемый сегментный регистр. Еще раз отметим, что во всех базовых и индексных способах адресации операндом является содержимое ячейки памяти, адрес которой находится в том или ином регистре или вычисляется сложением содержимого двух регистров.

Примеры:

```
mov    AL,[BX]    ;Сегментный адрес предполагается в DS, смещение в BX
mov    DL,ES:[BX] ;Сегментный адрес в ES, смещение в BX
mov    DX,[BP]    ;Сегментный адрес в SS, смещение в BP
mov    AL,[DI]    ;Сегментный адрес в DS, смещение в DI
```

Базовая и индексная адресации памяти со смещением

Относительный адрес операнда определяется суммой содержимого регистра (BX, BP, SI или DI) и указанного в команде числа, которое довольно неудачно называют смещением.

Пример:

```
mas db  1,2,5,3,7,9,8,3,4 ;Массив символов
        mov   BX,2        ;BX=индекс элемента в массиве
        mov   DL,mas[BX]  ;В DL заносится третий элемент массива
```

В этом примере относительный адрес адресуемого элемента массива mas вычисляется как сумма содержимого BX (2) и значения символического обозначения mas, которое совпадает с относительным адресом начала массива mas. В результате в регистр DL будет загружен элемент массива mas с индексом 2, т. е. число 5. Предполагается, что базовый адрес сегмента, в который входит массив mas, загружен в DS. Такой же результат даст такая последовательность команд:

```
mov   BX,offset mas;BX=относительный адрес ячейки mas
mov   DL,2[BX]
```

Здесь относительный адрес адресуемого элемента массива mas вычисляется как сумма содержимого регистра BX и дополнительного смещения, задаваемого константой 2. Последняя команда может быть записана в следующем виде:

```
mov   DL,[BX+2]
mov   DL,[BX]+2
```

Адресация с помощью регистров SI и DI осуществляется аналогично. При использовании регистра BP следует помнить, что в качестве сегментного регистра по умолчанию подразумевается регистр SS.

Базово-индексная адресация памяти

Относительный адрес операнда определяется суммой содержимого базового и индексного регистров. Допускается использование следующих пар:

```
[BX][SI]
[BX][DI]
[BP][SI]
[BP][DI]
```

Если в качестве базового регистра выступает BX, то в качестве сегментного подразумевается DS (первые две команды); при использовании в качестве базового регистра BP сегментным регистром по умолчанию назначается SS (вторые две команды). При необходимости можно явно указать требуемый сегментный регистр.

Примеры:

```
mov  BX,[BP][SI]      ;В BX засылается слово из стека (сегментный адрес в SS),
                      ;а смещение вычисляется как сумма содержимого BP и SI
mov  BX,ES:[BP][SI]   ;В BX засылается слово из сегмента, адрес которого
                      ;находится в ES, а смещение вычисляется как сумма
                      ;содержимого BP и SI
mov  ES:[BX+DI],AX    ;В ячейку памяти, сегментный адрес которой
                      ;хранится в ES, а смещение равно сумме содержимого
                      ;BX и DI, пересылается содержимое AX
```

Базово-индексная адресация памяти со смещением

Относительный адрес операнда определяется суммой трех величин: содержимого базового и индексного регистров, а также дополнительного смещения. Допускается использование тех же пар регистров, что и в базово-индексном способе; так же действуют и правила определения сегментных регистров.

Примеры:

```
mov  mas[BX][SI],10   ;Число 10 пересылается в ячейку памяти, сегментный
                      ;адрес которой хранится в DS, а смещение равно
                      ;сумме содержимого BX и SI и смещения ячейки mas
mov  AX,[BP+2+DI]     ;В AX пересылается из стека слово,
                      ;смещение которого равно сумме BP, DI и "добавки" 2
```

Значительная часть рассмотренных выше способов адресации служит для обращения к ячейкам памяти. Таким образом, один и тот же конечный результат можно получить с помощью различных способов адресации. Например, все три приведенные ниже команды:

```
mov   DL,mas+3
mov   DL,mas[BX]   ;В BX заранее занесено число 3
mov   DL,[SI][BX]  ;В BX заранее занесено число 3, а в SI - смещение mas
```

приведут к загрузке в регистр DL четвертого элемента массива mas (если выполняются описанные в комментариях условия). Однако команды с использованием различных способов адресации занимают различный объем памяти и выполняются за разное время. Так, первая из приведенных выше команд потребует для выполнения 15 машинных тактов, вто-

рая – 18, а третья – 16. Разница невелика, однако при многократном выполнении команд в циклах суммарный эффект может быть значителен. С другой стороны, первые две команды занимают в памяти по 4 байта, а третья – только 2. Таким образом, тщательный выбор способов адресации позволяет в какой-то степени оптимизировать программы по времени выполнения или требуемой памяти, а иногда и по тому и по другому.

Статья 11. **Числа и символы**

Практически любая программа содержит в себе данные, с которыми она работает. Отвлекаясь от назначения этих данных и смысла, который в них вкладывается программистом или пользователем программы, можно считать, что данные могут быть трех видов: числа, тексты и зарезервированные поля, подлежащие заполнению по ходу выполнения программы. Со всеми этими разновидностями данных мы уже сталкивались. Как правило, данные описываются в программе в сегменте данных, хотя, как мы увидим позже, в некоторых случаях оказывается удобным расположить данные в сегменте команд.

Для описания числовых данных используются главным образом три директивы ассемблера: db (define byte, определить байт) для записи байтов, dw (define word, определить слово) для записи слов и dd (define double, определить двойное слово) для записи двойных слов. Каждая директива позволяет записывать как одиночные (скалярные) данные, так и массивы, причем данные можно задавать в двоичной, десятичной или 16-ричной системах счисления:

```
endsym  db    26           ;Число 26 занимает 1 байт
sized   dw    257          ;Число 257 занимает 1 слово
sizeh   dw    201h         ;То же число указано в 16-ричной форме
sizeb   dw    100000001b   ;То же число указано в двоичной форме
array4  db    10,20,30,30  ;Байтовый массив из 4 членов
array4w dw    10,20,30,40  ;Массив слов из тех же чисел
mil40   dd    40000000     ;Число 40 миллионов в двойном слове
max     dd    0FFFFFFFFh   ;Максимально возможное целое число в двойном слове
```

Обратите внимание на последнее предложение. Если 16-ричное число начинается с изображения десятичной цифры, то оно просто завершается буквой h, например: 21h, 3A7h, 8FFFh. Если же число начинается с буквы (обозначающей 16-ричную цифру), то перед ним надо обязательно ставить 0, чтобы транслятор понял, что это именно число, а не имя ячейки: 0FFF8h, 0A1h, 0C000h.

Для резервирования места под массивы используется оператор dup (duplicate, дублировать), с помощью которого можно выделять память байтами, словами или двойными словами, заполняя се заданным числом:

```
escs    db 256 dup (27)    ;Массив из 256 байт, заполненных числом 27
nuls    dw 500 dup (0)     ;Массив из 500 слов, заполненных нулями
ffffs   dd 6 dup (3FFFFh)  ;Массив из 6 двойных слов с числами 3FFFFh
```

Если исходное содержимое объявляемого массива не имеет значения, то в скобках после оператора dup вместо числа можно указать знак вопроса:

```
stk     dw 256 dup (?)
```

Фрагменты текста (их обычно называют символьными строками) вводятся в программу с помощью того жс оператора db; текст при этом заключается в одинарные или двойные кавычки:

```
sym      db 'A'
msg      db '~~~Осторожно, злая собака!~~~'
pname    db "Иванов И.И."
stk      db 512 dup ('*')   ;Символьный массив
```

В последнем предложении массив объсмом 512 байт заполнен кодами знака звездочки.

Для того чтобы к данным можно было обращаться, они должны иметь имена. Имена данных могут включать латинские буквы, цифры (не в качестве первого знака имени) и некоторые специальные знаки, например знаки подчеркивания (_), доллара ($) и коммерческого at (@). Имена данных следует выбирать таким образом, чтобы они отражали назначение конкретного данного.

Присвоение данным символических имен позволяет обращаться к ним в программных предложениях, не заботясь о фактических адресах этих данных. Например, команда

```
         mov    DL,sym
```

занесет в регистр DL содержимое ячейки sym независимо от того, в каком месте сегмента данных эта ячейка определена и в какое место физической памяти она попала. Однако программист, использующий язык ассемблера, должен иметь отчетливое представление о том, каким образом назначаются адреса ячейкам программы, и уметь работать не только с символическими обозначениями, но и со значениями адресов. Для обсуждения этого вопроса рассмотрим пример сегмента данных, в котором определяются данные различных типов. Хотя пример носит абстрактный характер, приведенный набор данных характерен для программы, выполняющей чтение с диска файла с некоторыми данными. В левой колонке укажем смещения данных (в 16-ричной форме), вычисляемые относительно начала сегмента.

```
        data      segment
0000h   handle    dw   0
0002h   fname     db   'FILE.001'
000Ah   fsize     dw   512
000Ch   buf       dw   dup 512 (0)
040Ch   msg       db   'OK!'
        data      ends
```

Сегмент данных начинается с числового данного с именем handle (в которое в дальнейшем будет помещен дескриптор файла), описанного как слово. Очевидно, что его смещение равно нулю. Поскольку это данное занимает 2 байта, следующее за ним данное fname получило смещение 2. Данное fname описывает строку текста (конкретно – имя файла) длиной 8 символов и занимает в памяти столько же байтов, поэтому следующее данное fsize (размер файла) получило относительный адрес 2 + 8 = 10 = Ah. Далее в сегменте данных зарезервировано поле buf для чтения содержимого файла размером 512 слов (1024 = 400h байт). Относительный адрес этого поля составляет Ah + 2 = Ch. Наконец, вслед за массивом buf описана короткая строка текста 'OK!', имеющая смещение Ch + 400h = 40Ch. Поскольку эта строка занимает 3 байта, полный размер сегмента данных составит 40Fh = 1039 байт, т. е. несколько больше 1 Кбайт.

Ассемблер, начиная трансляцию сегмента (в данном случае сегмента данных), начинает отсчет его относительных адресов. Этот отсчет ведется в специальной переменной транслятора (не программы!), которая называется счетчиком текущего адреса и имеет символическое обозначение знака $. По мере обработки полей данных их символические имена сохраняются в создаваемой ассемблером таблице имен вместе с соответствующими им значениями счетчика текущего адреса. Другими словами, введенные нами символические имена получают значения, равные их смещениям. Таким об-

разом, с точки зрения транслятора handle = 0, fname = 2, fsize = Ah и т. д. Поэтому, например, команда

```
        mov   DX,offset fname
```

трактуется ассемблером как

```
        mov   DX,02h
```

и приводит к записи в регистр DX числа 2 (смещения строки fname).

Приведенные рассуждения приходится использовать при обращении к "внутренностям" объявленных данных. Пусть, например, мы по ходу программы решили вывести на экран вместо заданной в сегменте команд фразы "OK!" фразу "OK?". Для этого можно воспользоваться описанной в программе строкой msg, модифицировав в ней последний символ с помощью следующей команды:

```
        mov   msg+2, '?'
```

Здесь мы "вручную" определили смещение интересующего нас символа в строке, зная, что все данные размещаются ассемблером друг за другом в порядке их объявления в программе. При этом, какое бы значение ни получило имя msg, выражение msg + 2 всегда будет соответствовать последнему байту этой строки.

До сих пор речь шла о данных, которые, в сущности, являлись переменными, в том смысле, что под них выделялась память и их можно было модифицировать. Язык ассемблера позволяет также использовать константы, которые являются символическими обозначениями чисел и могут использоваться всюду в тексте программы как наглядные эквиваленты этих чисел:

```
datasize = 10000
        mov   CX,datasize
        mov   CX,10000
```

Последние две команды полностью эквивалентны.

При определении констант допустимо выполнение арифметических операций. Пусть нам надо задать позицию символа на экране. Учитывая, что каждый символ записывается в видеопамяти в 2 байтах (в первом – код ASCII символа, а во втором – его атрибут), строка экрана имеет длину 80 символов, а высота экрана составляет 25 строк, то для вывода некоторого символа в середину экрана его смещение в видеопамяти от начала видеостраницы можно определить следующим образом:

```
position=80*2*12+40*2
```

Такая запись достаточно наглядна, и ее легко модифицировать, если мы решим вывести символ в какую-то другую область экрана.

Константами удобно пользоваться для определения длины текстовых строк:

```
cmd      db 'Вводите команду: '
cmd_l = $-cmd
```

В этом примере константа cmd_l получает значение длины строки cmd (в данном случае 17 байт, включая символ пробела, завершающий это сообщение), которая вычисляется как разность значения счетчика текущего адреса после определения строки и ее начального адреса cmd. Такой способ удобен тем, что при изменении содержимого строки достаточно перетранслировать программу и та же константа cmd_l автоматически получит новое значение.

Рассматривая описание в программе символьных (текстовых) строк, мы считали, что каждый символ занимает в памяти 1 байт и, видимо, записывается в виде того или

иного числа. Однако символ – это все-таки не число, а изображение. Каким же образом символы записываются в памяти компьютера и как образуются на экране (или принтере) их изображения?

Код символа занимает 1 байт и, следовательно, может существовать всего 256 различных символов с кодами от 00h до FFh (от 0 до 255). Таблица соответствия кодов символов их изображениям называется кодовой страницей или таблицей кодов ASCII. Поскольку в разных странах используются разные алфавитные символы, существует много кодовых страниц с различными номерами. На рис. 11.1 приведена стандартная для нашей страны таблица кодов ASCII (с номером 866).

Из рисунка видно, что первая половина таблицы содержит, помимо некоторых специальных значков, цифры, латинские буквы (строчные и прописные) и некоторые общеупотребительные знаки (препинания, арифметические и пр.); во второй половине таблицы, начиная с кода 80h = 128, содержатся символы национального алфавита и псевдографики. Первая половина таблицы одинакова для всех кодовых страниц, вторая половина – для каждой страны своя.

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00		☺	☻	♥	♦	♣	♠	•	◘	○	◙	♂	♀	♪	♫	☼
10	►	◄	↕	‼	¶	§	▬	↨	↑	↓	→	←	∟	↔	▲	▼
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	⌂
80	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
90	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
A0	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
B0	░	▒	▓	│	┤	╡	╢	╖	╕	╣	║	╗	╝	╜	╛	┐
C0	└	┴	┬	├	─	┼	╞	╟	╚	╔	╩	╦	╠	═	╬	╧
D0	╨	╤	╥	╙	╘	╒	╓	╫	╪	┘	┌	█	▄	▌	▐	▀
E0	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
F0	Ё	ё	»	«	„	“	÷	≈	°	·	·	√	№	¤	■	

Рис. 11.1. Таблица кодов символов

Строки символов, включаемые в программу, преобразуются транслятором в последовательность соответствующих этим символам кодов ASCII. Например, строка

```
msg     db 'OK!'
```

транслируется в последовательность трех кодов 4Fh, 4Bh и 21h, которые можно обнаружить в листинге трансляции или в окне дампа отладчика. Таким образом, в памяти компьютера, так же как и в файлах на дисках, символьные строки хранятся в виде последовательностей чисел (кодов ASCII). При выводе текста на экран в видеопамять компьютера также посылаются числа. Видеосистема компьютера преобразует эти числа в требуемую последовательность точек, составляющих изображения символов. Правила отображения задаются с помощью специальных таблиц, хранящихся в системных файлах EGA.CPI, EGA2.CPI и EGA3.CPI (в последний файл как раз и входит русскоязычная кодовая страница 866), а также в программах русификаторов. В процессе начальной загрузки компьютера требуемая таблица загружается в память знакогенератора видеосистемы, чем и устанавливается соответствие кодов изображениям конкретных символов.

Имея перед собой кодовую таблицу (см. рис. 11.1), легко вывести на экран любую последовательность включенных в нее символов, в том числе и тех, которые отсутствуют на стандартной клавиатуре. Например, посылка на экран числа 10h приведет к

выводу сплошной стрелки с острием вправо, а число 11h выведет такую же стрелку с острием влево. Следует, однако, иметь в виду, что некоторые из кодов верхних двух строк рис. 11.1 воспринимаются системными программами MS-DOS как управляющие. Например, код 7 приведет к кратковременному включению звукового сигнала (гудка), код 0Ah – к переводу строки, а код 26h может вообще прервать вывод на экран. Поэтому лучше использовать только коды в диапазоне 20h...0FEh (32...254). С другой стороны, для перевода курсора на экране терминала на следующую строку необходимо послать на экран пару кодов 0Dh и 0Ah.

В завершение этого раздела рассмотрим простую программу (пример 11.1), выводящую в середину пустого экрана текст в рамке. Поскольку в DOS отсутствуют средства позиционирования курсора или очистки экрана, для образования опрятного кадра придется воспользоваться длинными последовательностями пробелов. Вспомним еще раз, что длина строки экрана составляет 80 символов и на экране помещается 25 строк.

Пример 11.1. Вывод на экран текста в обрамлении

```
;В сегменте команд
        mov    AH,09h
        mov    DX,offset msg
        int    21h
;В сегменте данных
msg db 10*80 dup (20h) ;Прокрутка 10 строк экрана
;Вывод трех строк (пробелы, обрамление, текст, пробелы)
db 35 dup (20h),0C9h,9 dup (0CDh),0BBh,34 dup (20h)
db 35 dup (20h),0BAh,10h,'ОСТАНОВ',11h,0BAh,34 dup (20h)
db 35 dup (20h),0C8h,9 dup (0CDh),0BCh,34 dup (20h)
db 11*80 dup (20h) ;Прокрутка еще 11 строк экрана
db '$';Завершающий знак доллара для функции 09h
```

Статья 12. **Esc-последовательности**

Выше уже отмечалось, что возможности DOS по управлению экраном весьма скудны. С помощью функции 09h можно выводить на экран символьные строки, включая в них управляющие символы и символы псевдографики. Позиционирование курсора, как и очистку экрана, приходится выполнять с помощью пробелов. В DOS нет никаких функций, с помощью которых можно было бы программно установить курсор в заданное место экрана; нельзя также изменить цвет выводимых символов.

Включение в систему устанавливаемого драйвера терминала (файл ANSI.SYS, поставляемый вместе с MS-DOS) дает пользователю дополнительные возможности управления экраном и клавиатурой. Для установки ANSI-драйвера в файл CONFIG.SYS следует включить такую строку:

```
DEVICEHIGH=C:\DOS\ANSI.SYS
```

(где C:\DOS – путь к каталогу, в котором находится файл ANSI.SYS).

Если в символьной строке, выводимой на экран, встречается код клавиши Esc (27 или 1Bh), за которым следует символ [, то ANSI-драйвер перехватывает последующие символы и интерпретирует их как команды управления экраном или клавиатурой. С помощью Esc-последовательностей можно очищать экран, перемещать по нему курсор, выбирать цвета фона и символа, изменять видеорежим, а также переопределять клавиши клавиатуры.

В дальнейших примерах будут использоваться такие Esc-последовательности:

Esc[2J – очистка экрана и перемещение курсора в левый верхний угол;

Esc[*строка*;*позиция*H – установка позиции курсора. Параметр *строка* обозначает y-координату курсора в пределах 1...25, параметр *позиция* – x-координату в пределах 1...80 (для видеорежима 80x25 символов);

Esc[*код1*;*код2*;*код3*m – выбор атрибутов символов. Параметры *код1*, *код2* и *код3* могут принимать значения, приведенные в табл. 12.1.

Таблица 12.1. Назначение кодов задания атрибутов символов

Код	*Назначение*	*Код*	*Назначение*
0	Нормальное изображение (белые символы на черном фоне)		
1	Выделение яркостью		
5	Выделение мерцанием		
30	Черные символы	40	Черный фон
31	Красные символы	41	Красный фон
32	Зеленые символы	42	Зеленый фон
33	Коричневые символы	43	Коричневый фон
34	Синие символы	44	Синий фон
35	Фиолетовые символы	45	Фиолетовый фон
36	Бирюзовые символы	46	Бирюзовый фон
37	Белые символы	47	Белый фон

Помимо перечисленных, имеются Esc-последовательности, служащие для выбора видеорежима, переопределения клавиш клавиатуры и др.

Модифицируем текст выводимой строки программы 11.1, включив в него Esc-последовательности управления экраном (пример 12.1).

Пример 12.1. Управление экраном с помощью Esc-последовательностей

```
;В сегменте команд
        mov    AH,09h
        mov    DX,offset msg
        int    21h
;В сегменте данных
msg     db 27,'[2J'        ;Очистка экрана
db 27,'[31;47m'            ;Задание цвета (красный по белому)
db 27,'[10;35H',0C9h,9 dup(0CDh),0BBh ;Позиционирование и символы
db 27,'[11;35H',0BAh,10h,'ОСТАНОВ',11h,0BAh ;Позиционирование и символы
db 27,'[12;35H',0C8h,9 dup(0CDh),0BCh ;Позиционирование и символы
db 27,'[0m',27,'[25;1H$'   ;Отмена цвета и позиционирование
```

Esc-последовательности для управления выводом на экран в настоящее время почти потеряли свое значение. В коммерческих программах, предназначенных для выполнения под управлением DOS, вывод на экран цветных символов и позиционирование курсора осуществляется либо с помощью средств BIOS, либо путем прямого программирования видеопамяти. Однако в домашней работе при проведении всякого рода исследований (изучение средств реального и защищенного режимов, исследование операционных систем, отладка программных алгоритмов и пр.) применение Esc-последовательностей очень удобно, так как позволяет без особого труда сделать вывод на экран более наглядным и приятным для работы. В дальнейших примерах книги мы будем иногда использовать это средство.

Статья 13. **Преобразование чисел в символьную форму**

Почти непременным атрибутом любой программы является вывод на экран не только текстовой, но и числовой информации. Если вывод текстов осуществляется весьма просто – достаточно описать выводимый текст с помощью оператора db и использовать затем подходящую функцию DOS (например, 09h или 40h), то для вывода числа его необходимо сначала преобразовать в символьную форму. Действительно, если послать на экран, например, код 5, мы увидим изображение трефового туза; чтобы получить на экране цифру 5, надо послать код ASCII этого символа, т. е. число 35h. Таким образом, для вывода числа необходимо каждую цифру этого числа заменить кодом ASCII ее изображения. Например, для вывода на экран десятичного числа 13508 следует послать на экран последовательность кодов 31h, 33h, 35h, 30h, 38h (см. рис. 11.1). При выводе на экран 16-ричных чисел в последовательность кодов цифр потребуется включать и коды латинских букв, например изображение числа 87AFh потребует посылки на экран последовательности кодов 38h, 37h, 41h, 46h (и, возможно, 68h, если мы захотим завершить изображение числа на экране символом h – признаком его 16-ричности).

Дело, однако, осложняется тем, что в памяти компьютера числа хранятся не в десятичной или 16-ричной форме, а в двоичной, в виде последовательности единиц и нулей. Поэтому алгоритм преобразования произвольного числа в символьную форму оказывается довольно громоздким. С другой стороны, при программировании на языке ассемблера другого способа вывода на экран чисел не существует и любой программист должен иметь под рукой отлаженные процедуры преобразования такого рода, чтобы иметь возможность по мере необходимости включать их в свои программы (например, в виде подпрограмм).

Двоичное число, находящееся в ячейке памяти или регистре, можно вывести на экран как в 16-ричной, так и в десятичной форме. При выводе результатов каких-либо вычислений, например обработки числовых данных, естественно использовать десятичную форму. Однако в работе программиста-исследователя удобнее иметь дело с 16-ричным представлением чисел, которое нагляднее отображает физическое содержимое ячеек и регистров. Рассмотрим возможный алгоритм такого преобразования.

Оформим программный фрагмент преобразования двоичного числа в символьную форму в виде процедуры-подпрограммы. Возможности подпрограмм и средства работы с ними будут подробнее описаны в разделе 2 этой книги, однако незнание тонкостей выполнения подпрограмм не должно помешать нам использовать их уже сейчас в прикладных целях. В простейшем случае достаточно знать, что подпрограммы оформляются в виде процедур и вызываются с помощью команды call с указанием имени вызываемой процедуры. Каждая подпрограмма должна завершаться командой ret, осуществляющей возврат из подпрограммы назад в вызывающую программу с передачей управления на команду, непосредственно следующую за командой call.

При выделении всех алгоритмов преобразования в отдельные процедуры-подпрограммы текст главной процедуры оказывается очень простым (пример 13.1). В регистр AX заносится преобразуемое число (предложение 1), в регистр SI – смещение поля string, в которое будет помещен результат преобразования, и командой call вызывается

подпрограмма wrd_asc преобразования 16-битового слова в символьную форму. После этого остается лишь вызовом функции 09h DOS вывести строку string на экран.

Строка string (предложение 47) содержит 4 пустых байта, за которыми следуют знаки h (для придания выводимому числу большего благообразия) и $ (как завершающий символ для функции 9).

Пример 13.1. Преобразование двоичного числа в 16-ричную символьную форму

```
;В сегменте команд
;Фрагмент главной процедуры
        mov     AX,16385      ;(1)Преобразуемое число
        mov     SI,offset string;(2)Адрес результата
        call    wrd_asc       ;(3)Вызов подпрограммы wrd_asc
        mov     AH,09h        ;(4)Функция вывода на экран
        mov     DX,offset string;(5)
        int     21h           ;(6)Вызов DOS
;Подпрограмма преобразования слова
;При вызове преобразуемое число в AX, DS:SI = адрес поля для результата
wrd_asc proc                  ;(7)Начало процедуры
        push    AX            ;(8)Сохраним исходное число в стеке
        and     AX,0F000h     ;(9)Выделим четверку битов
        mov     CL,12         ;(10)Сдвинем ее на 12 бит
        shr     AX,CL         ;(11)в начало AL
        call    bin_asc       ;(12)Преобразуем в символ ASCII
        mov     byte ptr [SI],AL;(13)Отправим в строку результата
        pop     AX            ;(14)Вернем исходное число
        push    AX            ;(15)И сохраним его снова
        and     AX,0F00h      ;(16)Выделим четверку битов
        mov     CL,8          ;(17)Сдвинем ее на 8 бит
        shr     AX,CL         ;(18)в начало AL
        inc     SI            ;(19)Сдвинемся по строке результата
        call    bin_asc       ;(20)Преобразуем в символ ASCII
        mov     byte ptr [SI],AL;(21)Отправим в строку результата
        pop     AX            ;(22)Вернем исходное число
        push    AX            ;(23)И сохраним его снова
        and     AX,0F0h       ;(24)Выделим четверку битов
        mov     CL,4          ;(25)Сдвинем ее на 4 бита
        shr     AX,CL         ;(26)в начало AL
        inc     SI            ;(27)Сдвинемся по строке результата
        call    bin_asc       ;(28)Преобразуем в символ ASCII
        mov     byte ptr [SI],AL;(29)Отправим в строку результата
        pop     AX            ;(30)Вернем исходное число
        push    AX            ;(31)И сохраним его снова
        and     AX,0Fh        ;(32)Выделим четверку битов
        inc     SI            ;(33)Сдвинемся по строке результата
        call    bin_asc       ;(34)Преобразуем в символ ASCII
        mov     byte ptr [SI],AL;(35)Отправим в строку результата
        pop     AX            ;(36)Восстановим AX и стек
        ret                   ;(37)Возврат в вызывающую процедуру
wrd_asc endp                  ;(38)Конец процедуры
;Подпрограмма преобразования 16-ричной цифры
;Преобразуемая четверка битов в младшей половине AL, результат в AL
bin_asc proc                  ;(39)Начало процедуры
        cmp     AL,9          ;(40)Цифра > 9
        ja      lettr         ;(41)Да, на преобразование в букву
        add     AL,30h        ;(42)Нет, преобразуем в символ 0...9
        jmp     ok            ;(43)И на выход из подпрограммы
lettr:  add     AL,37h        ;(44)Преобразуем в символ A...F
ok:     ret                   ;(45)Возврат в вызывающую процедуру
bin_asc endp                  ;(46)Конец процедуры
;В сегменте данных
string  db      4 dup ('?'),'h$';(47)
```

Рассмотрим сначала операцию преобразования в символ четверки битов, которая соответствует одной 16-ричной цифре. В машинном слове 4 таких цифры. Мы должны, в зависимости от содержимого каждой четверки двоичных разрядов, получать код ASCII соответствующей 16-ричной цифры от 0 до F. Поскольку эту операцию придется выполнять для каждой четверки битов исходного числа, т. е. 4 раза, удобно оформить ее в виде вложенной подпрограммы (процедура bin_asc), которая будет вызываться из процедуры wrd_asc.

На рис. 13.1 изображены 4-битовые двоичные числа, их представление в 16-ричной форме, а также коды ASCII символов, отображающих 16-ричное представление этих чисел. Видно, что для первых 10 16-ричных цифр их преобразование в символы требует прибавления к числу 30h (кода символа 0); для последних шести цифр добавка составит 37h. Этим и определяется алгоритм преобразования.

а	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
б	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
в	30h	31h	32h	33h	34h	35h	36h	37h	38h	39h	41h	42h	43h	44h	45h	46h

Рис. 13.1. 4-битовые числа в двоичной (а) и 16-ричной (б) формах и коды ASCII изображений 16-ричных цифр (в)

Анализируемое число (в регистре AL) сравнивается с числом 9 (предложение 40). Если содержимое анализируемого 4-битового поля больше девяти, т. е. должно изображаться буквой, командой ja (jump if above, переход, если выше), происходит переход на метку lettr, где к коду числа прибавляется 37h. В противном случае число должно отображаться цифрой и к его коду прибавляется 30h (предложение 42). Таким образом, после возврата из подпрограммы bin_asc в регистре AL содержится код ASCII 4-битового числа в 16-ричном представлении.

Обратимся теперь к подпрограмме wrd_asc, выполняющей преобразование 16-битового слова. При вызове подпрограммы преобразуемое число должно быть помещено в регистр AX. Поскольку в процессе преобразования числа содержимое AX разрушается, исходное число прежде всего сохраняется в стеке (предложение 8). Для выделения старшей четверки битов над содержимым регистра AX выполняется операция логического умножения на число 0F000h (предложение 9). Команда логического умножения and обнуляет все биты первого операнда, в данном случае AX, соответствующие сброшенным битам ее второго операнда (у нас числа 0F000h), и оставляет без изменения остальные биты первого операнда (рис. 13.2).

Исходное число	9	2	C	4h
Оно же	1001	0010	1100	0100
Операнд-маска	1111	0000	0000	0000
Результат в AX	1001	0000	0000	0000

Рис. 13.2. Действие команды and

Мы выделили старшую четверку битов исходного числа, однако она находится в самом старшем полубайте регистра AX, в то время как для правильной работы подпрограммы bin_asc преобразуемая четверка должна располагаться в младшем полубайте AL. Команда логического сдвига вправо shr (shift right, сдвиг вправо) (предложение 11) сдвигает вправо все содержимое с помощью регистра CL. "Выпавшие" с правого конца регистра биты теряются (строго говоря, последний

выпавший бит сохраняется в флаге CF, но для данной программы это значения не имеет), а освобождающиеся биты с левой стороны операнда заполняются двоичными нулями. Сдвинув старший полубайт AX вправо на 12 разрядов, мы переместили его на место младшего полубайта AL, что и требуется для подпрограммы bin_asc. Теперь можно вызвать эту подпрограмму (предложение 12) и, получив из нее код ASCII первой цифры, записать ее в выделенное заранее поле для результата преобразования, смещение которого по условиям вызова подпрограммы wrd_asc должно находиться в регистре SI (предложение 13).

В предложениях 14...21 выполняется обработка следующей четверки битов. Сначала в регистр AX из стека выталкивается его исходное содержимое, которое тут же снова сохраняется в стеке. Далее выделяется вторая слева четверка битов, которой соответствует операнд 0F00h команды and (предложение 16) и которая сдвигается уже не на 12, а на 8 разрядов (предложения 17 и 18). Результат преобразования этой 16-ричной цифры должен быть записан в следующий байт выходной строки, поэтому командой inc указатель SI получает приращение 1. Вызовом процедуры bin_asc и переносом возвращаемого ею значения в строку string этот фрагмент завершается.

В предложениях 22...29 аналогично выполняется обработка третьей слева четверки битов, в предложениях 30...35 – самой правой четверки. Далее восстанавливается содержимое регистра AX (предложение 36), после чего командой ret управление возвращается в вызывающую программу.

На протяжении этой книги нам не раз придется обращаться к операции преобразования 16-битового (или 32-битового) слова в символьную строку с последующим выводом его на экран с целью визуального контроля его содержимого. Учитывая важность этой операции в практической работе на языке ассемблера, попробуем оптимизировать приведенные выше процедуры. Что касается процедуры bin_asc, то она слишком проста, чтобы ее можно было как-то улучшить. Иначе дело обстоит с процедурой wrd_asc, в которой много повторений. Фактически основная часть ее алгоритма повторяется 4 раза с небольшими изменениями. Естественно поместить этот фрагмент в цикл из четырех шагов, предусмотрев в нем настройку двух изменяемых значений: маски для выделения битов и числа побитовых сдвигов.

При модификации примера 13.1 оказалось удобным использовать в нем дополнительные команды, реализованные впервые в процессорах 80386. Для того чтобы ассемблер распознавал эти команды, в программе должна присутствовать директива .386. Однако, как уже отмечалось в статье 8, в этом случае ассемблер создаст 32-разрядное приложение, которое не будет работать в операционной системе MS-DOS. Для того чтобы приложение осталось 16-разрядным, директивы segment для сегментов команд и данных должны быть дополненными описателями use16:

```
.386                    ;Будут дополнительные команды 80386
text segment use16      ;Начало сегмента команд
...
data segment use16      ;Начало сегмента данных
```

Дополнительные описатели не требуют, чтобы в программе обязательно использовались новые команды процессора. Поэтому все последующие примеры этой книги, независимо от использования в них процедуры преобразования числа в символьную

строку, можно отлаживать в приведенном выше варианте. Рассмотрим теперь модифицированный вариант процедуры wrd_asc (пример 13.2).

Пример 13.2. Оптимизированная процедура преобразования двоичного числа в символьную форму

```
wrd_asc proc                ;(1)
        pusha               ;(2)Сохраним все регистры
        mov   BX,0F000h     ;(3)В BX будет маска битов
        mov   DL,12         ;(4)В DL будет число сдвигов AX
        mov   CX,4          ;(5)Счетчик цикла
cccc:   push  CX            ;(6)Сохраним его
        push  AX            ;(7)Сохраним исходное число в стеке
        and   AX,BX         ;(8)Выделим четверку битов
        mov   CL,DL         ;(9)Отправим в CL число сдвигов
        shr   AX,CL         ;(10)Сдвинем на CL бит вправо
        call  bin_asc       ;(11)Преобразуем в символ ASCII
        mov   byte ptr [SI],AL;(12)Отправим в строку результата
        inc   SI            ;(13)Сдвинемся по строке вправо
        pop   AX            ;(14)Вернем в AX исходное число
        shr   BX,4          ;(15)Модифицируем маску битов
        sub   DL,4          ;(16)Модифицируем число сдвигов
        pop   CX            ;(18)Восстановим счетчик цикла
        loop  cccc          ;(19)Цикл
        popa                ;(20)Восстановим все регистры
        ret                 ;(21)Возврат из подпрограммы
wrd_asc endp                ;(22)
```

Новый вариант процедуры использует в качестве мест временного хранения внутренних данных регистры процессора BX, CX и DL. Если иметь в виду применение этой подпрограммы в качестве стандартного инструментального средства, то ее необходимо дополнить командами сохранения используемых регистров в начале процедуры и восстановления их в конце, чтобы при обращении к подпрограмме не нарушалось содержимое этих регистров, возможно, используемых в вызывающей программе. Проще всего сохранить все регистры командой pusha (предложение 2) и восстановить их ответной командой popa (предложение 20).

В начале процедуры в регистр BX заносится начальное значение маски (0F000h) для выделения старшей четверки битов, в регистр DL – начальное значение числа побитовых сдвигов исходного числа для перемещения выделенной четверки битов в регистр AL, а в регистр CX – число шагов цикла. Однако регистр CX будет нужен для задания числа сдвигов в команде shr, поэтому в первом же предложении (6) тела цикла его содержимое сохраняется в стеке. Операции преобразования четверки битов в символ (предложения 8...14) выполняются практически так же, как и в предыдущем варианте. Далее сдвигом содержимого регистра BX вправо на 4 бита выполняется модификация маски (значение маски 0F000h заменяется на 0F00h), а вычитанием 4 из регистра DL изменяется число битов сдвига выделенной четверки битов (предложения 15 и 16). Наконец, восстанавливается счетчик цикла и выполняется команда loop возврата на метку cccc. После выполнения четырех шагов процедура завершается командой ret.

В приведенном варианте заметно сократился исходный текст процедуры (22 предложения вместо 32) и несколько уменьшился размер выполнимого модуля. Именно в таком виде процедуры bin_asc и wrd_asc будут использоваться в дальнейших примерах этой книги.

Статья 14. **Динамическое исследование программ**

Разработанную нами программу вывода на экран символьных эквивалентов 16-ричных чисел можно использовать для получения динамического дампа интересующих нас ячеек памяти или регистров в процессе выполнения программы. Пусть, например, мы хотим узнать, где в памяти находится наша программа.

Любая программа, загруженная в память, включает три важных для программиста компонента: окружение, префикс программы PSP и собственно программу, которая (в случае программы типа .EXE) может состоять из нескольких сегментов. Поскольку окружение и сама программа (включая PSP) рассматриваются DOS как отдельные блоки памяти, и та и другая структура предваряются блоками управления памяти (Memory Control Block, MCB) размером 16 байт. С помощью этих блоков DOS ведет учет свободной и занятой памяти (рис. 14.1).

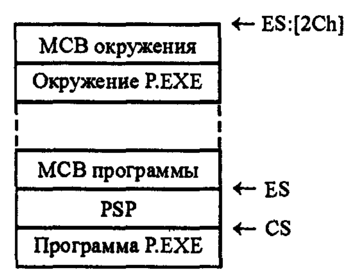


Рис. 14.1. Память, занимаемая программой

Окружение представляет собой область памяти, в которой в виде символьных строк записаны значения переменных, называемых переменными окружения, например PROMPT= pg. Здесь PROMPT – переменная окружения, а pg – ее конкретное значение, которое может быть и другим.

К вопросам использования блоков управления памятью и окружения мы вернемся в 3-м разделе этой книги. Здесь мы рассмотрим программу, которая после запуска выводит на экран терминала сегментные адреса окружения, префикса программного сегмента и сегмента команд (пример 14.1).

После загрузки программы в память регистры ES и DS указывают на PSP, регистр CS – на сегмент команд, а в слове со смещением 2Ch от начала PSP содержится сегментный адрес окружения программы.

Пример 14.1. Динамический вывод на экран характерных сегментных адресов

```
main    proc
        mov    AX,data
        mov    DS,AX
;Получим адрес окружения (находится по адресу 2Ch от начала PSP)
        mov    AX,ES:[2Ch]
        mov    SI,offset envir+10;Смещение приемника
        call   wrd_asc      ;Преобразование адреса окружения в символы
;Получим адрес PSP. Он находится в ES
        mov    AX,ES
        mov    SI,offset psp+10;Смещение приемника
        call   wrd_asc      ;Преобразование адреса PSP в символы
;Получим адрес сегмента команд. Он находится в CS
        mov    AX,CS
        mov    SI,offset csseg+10;Смещение приемника
        call   wrd_asc      ;Преобразование адреса сегмента команд в символы
```

```
;Выведем сформированный текст на экран
        mov     AH,09h
        mov     DX,offset envir
        int     21h
        ...                     ;Завершим программу
main    endp
wrd_asc proc
        ...
wrd_asc endp
bin_asc_proc
        ...
bin_asc endp
;Поля данных
envir   db      'Окружение=****h',13,10
psp     db      'Префикс=  ****h',13,10
csseg   db      'Программа=****h',13,10,'$'
```

Для повышения наглядности вывода в программе предусмотрены три отдельные символьные строки для выводимых сегментных адресов. Каждая строка начинается с поясняющего текста. Поля строк, предназначенные для хранения преобразованных адресов, начинаются с байта 10 от начала каждой строки. Последняя строка завершается знаком $.

Главная процедура состоит из трех практически одинаковых участков, в которых выполняется занесение в регистр AX преобразуемой величины, настройка регистра SI на начало поля для результата преобразования и вызов подпрограммы wrd_asc для преобразования сегментного адреса в символьную форму с занесением результата в символьную строку.

Сформировав поле данных, программа с помощью функции DOS 09h выводит его на экран.

Обратите внимание на команду mov AX,ES:[2Ch]. В ней указано не символическое имя адресуемой ячейки (сегмент с PSP формирует система, и у его ячеек нет никаких имен), а известное нам числовое смещение 2Ch от начала PSP. Кроме того, явно указано, что адрссоваться надо к сегменту, адрес которого находится в ES. Такой способ записи адреса, когда вместо подразумеваемого по умолчанию сегментного регистра DS явно указывается другой сегментный регистр, используется в практическом программировании очень часто и носит специальное название замены сегмента.

Статья 15. **Знаковые и беззнаковые числа операции**

Выше ужс отмечалось, что в машинное слово или регистр можно записать 64 К разных чисел в диапазоне от 0000h до FFFFh, или от 00000 до 65535. Это справедливо в том случас, если мы все возможныс числа рассматриваем как положительные (беззнаковыс). Если, однако, мы хотим в какой-то программе работать как с положительными, так и с отрицательными числами, т. е. с числами со знаком, нам придется часть чисел из их полного диапазона (наиболес естественно – половину) считать отрицательными. В вычислительной технике принято отрицательными считать все числа, у которых установлен старший бит, т. с. числа в диапазоне 8000h...FFFFh. Положительными же считаются числа со сброшенным старшим битом, т. е. числа в диапазоне 0000h...7FFFh. При этом отрицательныс числа записываются в дополнительном коде, который образуется из прямого путем замены всех нулей единицами и наоборот (обратный код) и прибавления к полученному числу единицы (рис. 15.1).

	Для байта	Для слова
Прямой код числа 5:	0000 0101	0000 0000 0000 0101
Обратный код числа 5:	1111 1010	1111 1111 1111 1010
	+1	+1
Дополннтельный код числа 5:	1111 1011	1111 1111 1111 1011

Рис. 15.1. Образование отрицательного числа

Следует подчеркнуть, что знак числа условен. Одно и то же число FFFBh, изображенное в нижней строке рис. 15.1, можно в одном контексте рассматривать как положительное (+65531), а в другом – как отрицательное (–5). Таким образом, знак числа является характеристикой не самого числа, а способа его обработки.

На рис. 15.2 представлена выборочная таблица 16-битовых чисел с указанием их знаковых и беззнаковых значений.

16-ричное представленне	Десятичное представление: без знака	со зиаком	
0000h	00000	00000	Ноль
0001h	00001	+00001	Минимальное положительное число
0002h	00002	+00002	
0003h	00003	+00003	Диапазон положительных чисел
. . .	. . .	. . .	
7FFEh	32766	+32766	
7FFFh	32767	+32767	Максимальное положительное число
8000h	32768	-32768	Макснмальное отрнцательное число
8001h	32769	-32767	
. . .	. . .	. . .	
FFFDh	65533	-00003	Диапазон отрнцательных чнсел
FFFEh	65534	-00002	
FFFFh	65535	-00001	Мннимальное отрицательное число

Рис. 15.2. Представление знаковых и беззнаковых чисел в 16-разрядном компьютере

Процессор может выполнять операции не только над словами, но и над байтами. Как и в случае целых слов, число в байте можно рассматривать как беззнаковое, и тогда оно может принимать значения от 000 до 255, или как число со знаком, и тогда диапазон положительных значений уменьшается в два раза (от 000 до 127), но возникает возможность записать столько же отрицательных чисел (от –001 до –128).

На рис. 15.3 представлена выборочная таблица байтовых (8-битовых) чисел с указанием их знаковых и беззнаковых значений.

16-ричное представление	Десятичное представление: без знака	со знаком	
00h	000	000	Ноль
01h	001	+001	Минимальное положительное число
02h	002	+002	
03h	003	+003	Диапазон положительных чисел
. . .	. . .	. . .	
7Eh	126	+126	
7Fh	127	+127	Максимальное положительное число
80h	128	-128	Максимальное отрицательное число
81h	129	-127	
82h	130	-126	Диапазон отрнцательных чисел
. . .	. . .	. . .	
FEh	254	-002	
FFh	255	-001	Мнннмальное отрицательное число

Рис. 15.3. Представление знаковых и беззнаковых байтовых чисел

При операциях с числами следует иметь в виду явление оборачивания, которое можно кратко выразить такими соотношениями:

```
FFFFh+0001h=0000h
0000h-0001h=FFFFh
```

Если последовательно увеличивать содержимое регистра или ячейки памяти, то, достигнув верхнего возможного предела FFFFh, число перейдет через эту границу, станет равным нулю и продолжит нарастать в области малых положительных чисел (1, 2, 3 и т. д.). Точно так же, если последовательно уменьшать некоторое положительное число, оно, достигнув нуля, перейдет в область отрицательных (или, что то же самое, больших беззнаковых) чисел, проходя значения 2, 1, 0, FFFFh, FFFEh и т. д.

Отсюда, между прочим, следует, что если при последовательном наращивании относительного адреса в сегменте данных (что обычно требуется при работе с массивами) перейти границу беззнакового представления чисел, то начнут адресоваться ячейки не за пределами нашего сегмента данных, а из самого его начала.

Среди команд процессора, выполняющих ту или иную обработку чисел, можно выделить команды, индифферентные к знаку числа (например, inc, dec, test), команды, предназначенные для обработки беззнаковых чисел (mul, div, ja, jb и др.), а также команды, специально рассчитанные на обработку чисел со знаком (imul, idiv, jg, jl и т. д.).

Рассмотрим в качестве примера команды умножения. Их две: mul (multiplication, умножение) для умножения беззнаковых чисел и imul (integer multiplication, целочисленное умножение) для работы со знаковыми числами. Результаты их выполнения при одних и тех же операндах могут радикально различаться.

Обе команды могут работать как со словами, так и с байтами. Они выполняют умножение числа, находящегося в регистрах AX (в случае умножения на слово) или AL (в случае умножения на байт), на операнд, который может находиться в каком-либо регистре или в ячейке памяти. Не допускается умножение на непосредственное значение, а также на содержимое сегментного регистра.

Размер произведения, т. е. число байтов в нем, всегда в два раза больше размера сомножителей. Для 1-байтовых операций полученное произведение записывается в регистр AX. Для 2-байтовых операций результат умножения, который имеет размер 32 бита, записывается в регистры DX:AX (в DX – старшая половина, в AX – младшая).

Рассмотрим несколько конкретных примеров действия команд знакового и беззнакового умножения.

```
mov   AL,3      ;Первый сомножитель=003
mov   BL,2      ;Второй сомножитель=002
mul   BL        ;AX=0006h=00006
mov   AL,3      ;Первый сомножитель=003
mov   BL,2      ;Второй сомножитель=002
imul  BL        ;AX=0006h=+00006
```

Обе команды, mul и imul, дают в данном случае одинаковый результат, поскольку знаковые положительные числа совпадают с беззнаковыми. Обратите внимание на то, что результат умножения, будучи в данном случае небольшим, занимает тем не менее весь регистр AX, затирая его старший байт.

```
mov   AL,0FFh   ;Первый сомножитель=0FFh=255
mov   BL,2      ;Второй сомножитель=002
mul   BL        ;AX=01FEh=00510
mov   AL,0FFh   ;Первый сомножитель=0FFh=-001
mov   BL,2      ;Второй сомножитель=002
imul  BL        ;AX=FFFEh=-00002
```

Здесь действие команд mul и imul над одними и теми же операндами дает разные результаты. В первом примере беззнаковое число FFh, которое интерпретируется как десятичное 255, умножается на 2, давая в результате беззнаковое 00510, или 01FEh. Во втором примере то же число FFh рассматривается как знаковое. В этом случае оно составляет –001. Умножение на 2 дает –002, или FFFEh.

Разная интерпретация одного и того же числа FFh обусловлена использованием в первом случае команды для обработки беззнаковых чисел, а во втором – знаковых.

Аналогичная ситуация возникает и при умножении целых слов:

```
mov   AX,0FFFFh   ;Первый сомножитель=0FFFFh=65535
mov   BL,2        ;Второй сомножитель=00002
mul   BL          ;DX:AX=0001h:FFFEh=0000131070
mov   AL,0FFFFh   ;Первый сомножитель=0FFFFh=-00001
mov   BL,2        ;Второй сомножитель=00002
imul  BL          ;DX:AX=FFFFh:FFFEh=-0000000002
```

В первом примере число без знака FFFFh (65535 в десятичном представлении) умножается на 2, давая в результате число без знака 0000131070, или 0001FFFEh. Это 32-битовое число размещается в двух регистрах. Старшая половина (0001h) записывается в регистр DX, затирая его предыдущее содержимое, младшая половина (FFFEh) в регистр AX. Во втором примере то же число FFFFh рассматривается как число со знаком. В этом случае оно составляет –00001. Умножение на 2 дает –0000000002, или FFFFFFFEh. По-прежнему старшая половина этого числа (FFFFh) записывается в DX, младшая половина (FFFEh) – в AX.

Другая важная группа команд, различающих числа со знаком и без знака, – это команды условных переходов. Эти команды позволяют осуществлять переходы на различные метки программы в зависимости от результата выполнения предыдущей команды. Команды условных переходов часто используют после команд сравнения (cmp), инкремента (inc), декремента (dec), сложения (add), вычитания (sub), проверки (test) и ряда других.

Приведем перечень команд условных переходов, чувствительных к "знаковости" числа.

Знаковые команды

jg (jump if greater – переход, если больше)
jge (jump if greater or equal – переход, если больше или равно)
jl (jump if less – переход, если меньше)
jle (jump if less or equal – переход, если меньше или равно)
jng (jump if not greater – переход, если не больше)
jnge (jump if not greater or equal – переход, если не больше и не равно)
jnl (jump if not less – переход, если не меньше)
jnle (jump if not less or equal – переход, если не меньше и не равно)

Беззнаковые команды

ja (jump if above – переход, если выше)
jae (jump if above or equal – переход, если выше или равно)
jb (jump if below – переход, если ниже)
jbe (jump if below or equal – переход, если ниже или равно)
jna (jump if not above – переход, если не выше)
jnae (jump if not above or equal – переход, если не выше и не равно)
jnb (jump if not below – переход, если не ниже)

Примеры команд, нечувствительных к знаку числа

je (jump if equal – переход, если равно)
jne (jump if not equal – переход, если не равно)
jc (jump if carry – переход, если флаг CF установлен)
jcxz (jump if CX=0 – переход, если CX=0)

Разница между знаковыми и беззнаковыми командами условных переходов заключается в том, что знаковые команды рассматривают понятия "больше-меньше" применительно к числовой оси –32 K...0...+32 K, а беззнаковые команды – применительно к числовой оси 0...64 K. Поэтому для команд знаковых переходов число 7FFFh (+32767) больше числа 8000h (–32768), а для команд беззнаковых 7FFFh (32767) – меньше чем 8000h (32768). Аналогично для знаковых команд 0 больше чем FFFFh (–1), а для беззнаковых – меньше.

Таким образом, при сравнении знаковых чисел используются термины "больше" и "меньше", а при сравнении беззнаковых – "выше" и "ниже".

Все сказанное справедливо и в том случае, когда команды условных переходов используются для анализа содержимого байтовых операндов. Так, для знаковых команд 7Fh (+127) больше, чем 80h (–128), а 0 больше, чем FFh (–1), а для беззнаковых команд – наоборот.

Статья 16. **Строковые команды**

В составе команд процессора МП 86 имеется группа команд, предназначенных для операций со строками символов или чисел т. е., по существу, с массивами произвольных данных. Таких команд всего 5:

- movs – пересылка строки;
- cmps – сравнение строк;
- scas – поиск в строке заданного элемента (сканирование строки);
- lods – загрузка из строки регистров AX или AL;
- stos – запись элемента строки из регистров AX или AL.

Команды имеют общие черты: они выполняются процессором в предположении, что адрес строки-источника находится в регистрах DS:SI, а адрес строки-приемника в ES:DI; при однократном выполнении они обрабатывают только один элемент, а для обработки строки должны предваряться префиксом повторения; в процессе обработки элементов строки регистры SI и DI автоматически смещаются по строке вперед (если DF=0) или назад (если DF=1); каждая команда имеет модификации для работы с байтами или словами (например, movsb и movsw).

Рассмотрим сначала особенности использования строковых команд на простых формальных примерах.

А. Пересылка (копирование) строки байтов

```
;В сегменте данных
src     db 10,20,30,40,50  ;Строка-источник
dest    db 5 dup (?)       ;Строка-приемник
;В сегменте команд
        mov    AX,data     ;Обычная инициализация
        mov    DS,AX       ;регистра DS
        mov    SI,offset sourc ;DS:SI→src
        push   DS          ;DS в стек
```

```
        pop     ES            ;Из стека в ES, теперь ES=DS
        mov     DI,offset dest;ES:DI→dest
        cld                   ;DF=0, движение по строке вперед
        mov     CX,5          ;Число пересылаемых байтов
rep     movsb                 ;Собственно пересылка
```

В этом примере команда movsb использована с префиксом повторения rep (repeat, повторять), который фактически заставляет процессор выполнить команду movsb число раз, равное содержимому регистра CX, т. е. в данном случае переслать 5 байт.

Мы видим, что перед использованием команды movsb надо выполнить целый ряд предварительных действий: поместить в регистры DS и ES сегментные адреса источника и приемника, а в регистры SI и DI – их смещения; с помощью команды cld (clear direction, сбросить направление) сбросить флаг процессора DF, в регистр CX заслать число пересылаемых байтов. После этого одна команда movsb с префиксом повторения rep выполняет операцию копирования сколь угодно длинной последовательности байтов (практически до 32 Кбайт, если и источник и приемник находятся в одном сегменте) на новое место.

Б. Сравнение двух строк

```
;В сегменте данных
sourc   db      'MYFILE01.~AT' ;Сравниваемая строка
fname   db      'MYFILE01.DAT' ;Эталон
;В сегменте команд
        mov     AX,data
        mov     DS,AX
        mov     SI,offset sourc
        push    DS
        pop     ES
        mov     DI,offset fname
        cld
        mov     CX,12
repe    cmpsb
        jne     no
yes:    ...
no:     ...
```

Сравнение пары строк осуществляется с помощью команд cmps, cmpsb и cmpsw. Префикс repe (repeat while equal, повторять, пока равно), использованный в этом примере, заставляет процессор выполнять сравнение последовательных элементов строк, пока эти элементы равны, т. е. до тех пор, пока не обнаружится пара различающихся элементов. Если все элементы оказываются попарно одинаковыми, сравнение выполняется CX раз, а после завершения цикла сравнения флаг нуля ZF устанавливается в 1. Если же в какой-то паре элементы оказались разными, цикл сравнения заканчивается, а флаг ZF устанавливается в 0 (отсутствие равенства операндов). Команда jne (или jnz) позволяет проанализировать результат сравнения (в сущности, последней сравниваемой пары элсментов) и перейти на соответствующую метку при обнаружении неравенства строк. В случае равенства всех CX пар элементов команда jne не срабатывает и выполняется следующее за jne предложение программы.

В приведенном примере сравниваемые строки различаются в 10-й паре байтов, поэтому после сравнения 10-й пары (символов D и ~) выполнение команды cmpsb "досрочно" завершится со сброшенным флагом ZF и сработает команда jne, передав управление на метку no.

В. Сканирование строки

Команды сканирования (scas, scasb и scasw) могут использоваться двояко. С префиксом repe команды, просматривая элементы строки, пропускают все вхождения искомого элемента в начале строки, останавливаясь на первом элементе, отличным от исходного. С префиксом repne (repeat while not equal, повторять, пока не равно) команды, наоборот, позволяют найти первое вхождение исходного элемента. Поскольку анализируемая строка считается приемником, она адресуется через регистры ES:DI. Искомый символ должен находиться в регистрах AL или AX.

```
;В сегменте данных, адресуемых через ES
tail    db '    /S:3'
;В сегменте команд
        mov   DI,offset tail
        cld
        mov   AL,' '       ;Искомый символ - пробел
        mov   CX,80        ;Максимальная длина данной строки
repe    scasb              ;Сканирование байтов, пока равно
        je    blank        ;Если в строке одни пробелы
gotit:  ...                ;Символ найден, DI=tail+5
blank:  ...
```

В приведенном выше примере поиск осуществляется, пока байты строки равны искомому символу (пробелу). Команда scasb прекращает выполнение, как только она обнаруживает первый байт, отличный от пробела. При этом следует иметь в виду, что после "выхода" из команды scasb регистр DI указывает на элемент массива, следующий за последним проанализированным, т. е. в данном случае на символ 'S' (а не '/'). Если (как это скорее всего и будет) нам нужно знать положение в массиве первого элемента, отличного от искомого, содержимое DI следует уменьшить на единицу.

```
;В сегменте данных, адресуемом через ES
words   dw 10,100,234,183,16789,0,15644,...
;В сегменте команд
        mov   DI,offset words ;ES:DI→words
        cld                ;Сканирование вперед
        mov   AX,0         ;Искомое число
        mov   CX,1000      ;Количество слов в массиве
repne   scasw              ;Сканирование слов, пока не равно
        jne   zeros        ;Если в массиве нет нулей
gotit:  ...                ;Первый 0 найден, DI=words+12
zeros:  ...
```

В приведенном выше примере поиск осуществляется, пока элементы массива не равны искомому числу (0). Команда scasw прекращает выполнение, как только она обнаруживает искомый элемент. При этом следует иметь в виду, что после "выхода" из команды scasw регистр DI указывает на элемент массива, следующий за последним проанализированным, т. е. в данном случае на число 15 644. Если (как это скорее всего и будет) нам нужно знать положение в массиве искомого элемента, содержимое DI следует уменьшить на 2.

Г. Загрузка регистра из строки

Пусть в два последовательных байта по адресу limits поступают результаты вычислений или измерений, влияющие на дальнейший ход программы. Нормальное продолжение программы допустимо, если первый байт больше или равен 85h, а второй – меньше или равен 82h. Анализ последовательности байтов и соответствующие условные переходы удобно организовать с использованием команды lodsb:

```
;В сегменте данных, адресуемом через DS
limits  db 90h,81h          ;Анализируемые числа
;В сегменте команд
        mov    SI,offset limits ;DS:SI→limits
        cld                 ;Сканирование вперед
        lodsb               ;AL=90h, DS:SI→limits+1
        cmp    AL,85h       ;Анализ первого байта, AL=90h
        jb     less85h      ;Переход, если меньше 85h (у нас не меньше)
        lodsb               ;Загрузка второго байта, AL=81h
        cmp    AL,82h       ;Анализ второго байта
        ja     over82h      ;Переход, если больше 82h (у нас не больше)
range:  ...                 ;Нормальное выполнение (наш случай):
                            ;1-й байт >=85h, 2-й байт <=82h
less85h:...                 ;1-й байт не годится
over82h:...                 ;2-й байт не годится
```

Д. Запись из регистра элемента строки

```
;В сегменте данных, адресуемом через ES
array   dw 1024 (?)
;В сегменте команд
        mov    DI,offset array ;ES:DI→array
        cld                 ;Сканирование вперед
        mov    AX,0         ;Заполнитель массива
        mov    CX,1024      ;Коэффициент повторения
rep     stosw               ;Заполнение всего массива нулями
```

В приведенном фрагменте одной командой stosw с префиксом повторения rep массив array заполняется числом 0, что приводит к затиранию его (более ненужного) старого содержимого.

В заключение рассмотрим более содержательный пример использования одной из строковых команд, конкретно – команды сравнения, для защиты программы от несанкционированного доступа с помощью пароля.

Идея простейшей защиты программы от несанкционированного запуска заключается в том, что где-то в программе записывается ключевое слово-пароль и программа, начав работать, требует ввода этого слова с клавиатуры. Если пользователь ввел пароль правильно, программа продолжает работать. Если пароль введен неверно, программа завершается. Таким образом, программа должна сравнить введенное слово с хранящимся в ней и фиксировать совпадение или наличие хотя бы незначительной разницы. Рассмотрим программу ввода и анализа пароля.

Вообще говоря, ввод пароля можно осуществить любыми функциями DOS ввода с клавиатуры. Однако обычно пароль вводят одной из функций, не отображающих вводимые символы на экране. Таких функций две – 07h и 08h. Разница между ними заключается в том, что функция 08h, зафиксировав ввод пользователем сочетания Ctrl+C, аварийно завершает программу, функция же 07h к Ctrl+C нечувствительна. Поскольку обе эти функции вводят лишь один символ, для ввода пароля их надо использовать в цикле. Выход из цикла можно организовать по-разному: после ввода обусловленного числа символов, по нажатии клавиши Enter либо как-то иначе. В примере 16.1 ввод пароля заканчивается нажатием клавиши Enter.

Пример 16.1. Ввод пароля и сравнение символьных строк

```
;Выведем запрос prompt с помощью функции DOS 09h
        ...
;Введем пароль
        mov    BX,0         ;(1)Инициализируем индексный регистр
```

```
pass:   mov    AH,08h         ;(2)Функция ввода без эха
        int    21h            ;(3)Вызов DOS
        cmp    AL,13          ;(4)Enter?
        je     compr          ;(5)Да, на сравнение
        mov    string[BX],AL;(6)Нет, сохраним символ
        inc    BX             ;(7)Инкремент индекса
        jmp    pass           ;(8)Повторять до Enter
;Будем сравнивать строки
compr:  push   DS             ;(9)Настроим ES на наш
        pop    ES             ;(10)сегмент данных
        mov    SI,offset inpt;(11)Смещение строки ввода
        mov    DI,offset pass;(12)Смещение строки с паролем
        cld                   ;(13)Направление вперед
        mov    CX,BX          ;(14)Инициализируем счетчик цикла
repe    cmpsb                 ;(15)Сравнение
        jne    err            ;(16)Строки не равны
        ...                   ;Продолжение программы
err:    mov    AX,4C00h       ;(17)Завершение программы
        int    21h            ;(18)и выход в DOS
;Поля данных
pass    db 'LuckyStrike'   ;(19)Ожидаемый пароль
inpt    db 80 dup (?)      ;(20)Поле для ввода пароля
prompt  db 'Введите пароль >>$'     ;(21)Запрос
```

Работа программы начинается с вывода на экран запроса программы. В данном случае запрос имеет вид "Введите пароль>>". Далее инициализируется регистр BX, который будет использоваться в качестве индексного, и ставится запрос к DOS на ввод символа без эха (предложения 2 и 3). Введенный код сравнивается с кодом клавиши Enter (предложение 4), и в случае равенства кодов командой je выполняется переход на участок сравнения строк. Если же нажата любая другая клавиша, ее код ASCII заносится в поле inpt со смещением относительно начала этого поля на число байтов, равное содержимому регистра BX. Выполняется инкремент регистра BX и командой безусловного перехода jmp управление передается в точку pass на ввод следующего символа.

При нажатии клавиши Enter выполняется настройка регистров для выполнения операции сравнения. Первая строка адресуется через пару регистров DS:SI, вторая – через регистры ES:DI. В предложениях 11 и 12 в регистры SI и DI заносятся смещения сравниваемых строк, а в предложении 13 командой cld устанавливается направление сравнения – от начала строки к ее концу.

Для того чтобы сравнить целую строку, команда cmpsb предваряется префиксом повторения repe. В этом случае операция сравнения выполняется до тех пор, пока символы двух строк совпадают, но не более CX раз. Поэтому требуется еще настроить и регистр CX. В нашем случае в него заносится конечное содержимое регистра BX, т. е. длина введенной с клавиатуры строки.

Выход из цикла повторения команды cmpsb происходит либо после обнаружения несовпадающих байтов (и тогда ясно, что строки не совпадают), либо по исчерпании счетчика цикла CX. В последнем случае все байты строк, кроме последних, наверняка совпадают, однако результат сравнения последней пары байтов неясен. Таким образом, сама по себе команда cmpsb, выполняемая в цикле, не позволяет судить о результатах сравнения. После этой команды необходим анализ результата сравнения последней пары байтов одной из команд условного перехода. Использованная в программе команда jne в случае неравенства передает управление на метку err, где выполняется аварийное завершение программы. Если же строки полностью совпадают, начинается нормальное выполнение содержательной части программы.

Статья 17. **Ввод с клавиатуры десятичных чисел**

Все данные, поступающие в компьютер с клавиатуры или выводимые на экран терминала, рассматриваются как коды ASCII вводимых символов. Для вывода на экран числа из ячейки памяти или регистра его следует предварительно преобразовать в коды ASCII соответствующих цифр той системы счисления, в которой требуется отобразить число на экране (этот вопрос рассматривался в статье 13). Точно так же при вводе числа с клавиатуры полученные символы надо преобразовать в число (двоичное, поскольку любые данные хранятся в компьютере исключительно в двоичной форме). Очевидно, что процедура преобразования будет зависеть от того, какое число мы хотим набрать на клавиатуре – десятичное или 16-ричное. В настоящей статье будет рассмотрен ввод с клавиатуры десятичных чисел.

Помимо преобразования символьных кодов цифр в двоичные числа и объединения этих чисел с учетом весов отдельных десятичных разрядов, в программе должен быть предусмотрен анализ вводимых символов и исключение из входного потока кодов, не являющихся кодами десятичных цифр (такую операцию часто называют фильтрацией). Кроме того, следует предусмотреть какое-либо соглашение о способе завершения ввода каждого числа. Например, можно всегда вводить пяти десятичных цифр (с лидирующими нулями, если число имеет меньше 5 десятичных разрядов). Удобнее вводить число без лидирующих нулей, завершая ввод нажатием клавиши Enter или другой выделенной для этого клавиши. В соглашениях о правилах ввода (как говорят, в интерфейсе с программой) должна быть оговорена реакция программы на неправильно нажатую клавишу. Проще всего заставить программу игнорировать все нецифровые символы, хотя может быть предусмотрена и иная реакция, например вывод сообщения об ошибкс или подача звукового сигнала. Далее следует решить, надо ли отображать на экране случайно введенные нецифровые символы.

Программный фрагмент, приведенный в примере 17.1, осуществляет ввод с клавиатуры десятичных чисел с любым числом разрядов от 1 до 5. Ввод каждого числа завершается нажатием клавиши Enter. Вводимые символы проверяются на их принадлежность десятичному числу; нецифровые символы не воспринимаются и не отображаются на экране. Результат преобразования (который остается в регистре AX) после преобразования в символьную строку выводится на экран для контроля.

Пример 17.1. Ввод с клавиатуры десятичного числа

```
;Выведем на экран запрос, свидетельствующий об ожидании ввода
        mov     AH,02h          ;(1)Вывод
        mov     DL,'>'          ;(2)запроса
        int     21h             ;(3)
        mov     DI,0            ;(4)Очистим регистр для результата
;Будем вводить и анализировать символы
inpt:   mov     AH,08h          ;(5)Функция ввода символа без эха
        int     21h             ;(6)
        cmp     AL,13           ;(7)Нажата клавиша Enter?
        je      done            ;(8)Да, ввод числа закончен
        cmp     AL,'9'          ;(9)Цифровой символ?
        ja      inpt            ;(10)Нет! На повторный ввод
        cmp     AL,'0'          ;(11)Цифровой символ?
        jb      inpt            ;(12)Нет! На повторный ввод
;Введен очередной цифровой символ. Выведем его на экран
        mov     AH,02h          ;(13)Функция вывода символа
        mov     DL,AL           ;(14)Перенесем символ в DL
```

```
        int     21h             ;(15)(для функции 02h)
        sub     AL,'0'          ;(16)Преобразуем символ в двоичное число
        xor     AH,AH           ;(17)Обнулим AH
        mov     CX,AX           ;(18)Сохраним полученную цифру в CX
        mov     AX,DI           ;(19)Результат преобразования предыдущих цифр
        mov     BX,10           ;(20)Множитель 10
        mul     BX              ;(21)AX=предыдущий результат * 10
        add     AX,CX           ;(22)Добавим к старому числу новую цифру
        mov     DI,AX           ;(23)Сохраним в регистре DI
        jmp     inpt            ;(24)На ввод следующей цифры
done:   mov     AX,DI           ;(25)Загрузим результат в AX
;Преобразуем число в AX в символьную строку
        mov     SI,offset result+3;(26)Смещение для wrd_asc
        call    wrd_asc         ;(27)Преобразуем в символы
;Выведем результат на экран функцией DOS 09h
        ...
wrd_asc proc
        ...
wrd_asc endp
bin_asc proc
        ...
bin_asc endp
;В полях данных
result  db ' = ',5 dup (?)
```

Принцип преобразования вводимой строки символов в число заключается в следующем. Первая введенная цифра (старший десятичный разряд) преобразуется в двоичное число. После ввода и преобразования следующей цифры первое число умножается на 10 и к полученному произведению прибавляется введенное число. Далее этот процесс повторяется до нажатия клавиши Enter.

Программа прежде всего выводит на экран запрос в виде знака > (предложения 1...3). Перед началом ввода очищается регистр DI, где будет накапливаться результат ввода последовательных цифр исходного числа. Вызовом функции DOS 08h ставится запрос на ввод с клавиатуры одного символа без отображения его на экране (отображать мы будет только цифры). Функция 08h возвращает введенный символ в AL. Код ASCII нажатой клавиши сравнивается с кодом ASCII клавиши Enter (код 13, предложение 7). Если нажата эта клавиша, процесс ввода цифр заканчивается переходом на метку done. При любой другой клавише выполняется анализ введенного кода. Содержимое AL сравнивается с символьным представлением цифры 9 (предложение 9). Если введенный код больше '9', он не является цифрой и его следует отбросить. Эту ситуацию отрабатывает команда ja, осуществляя переход на начало блока ввода очередного символа. Если введенный код прошел проверку на верхнюю границу, он сравнивается с нижней границей – кодом '0' (предложение 11). Команда jb осуществляет переход на начало блока ввода символа, если введен символ с кодом меньше '0'. Следующее предложение 13 будет выполняться, только если нажата клавиша с цифрой. Функцией DOS 02h введенный символ выводится на экран (предложения 13...15), после чего начинается его преобразование в число.

В предложении 17 с помощью команды xor выполняется очистка старшего байта регистра AX. Вообще команда xor (exclusive or, исключающее ИЛИ) анализирует биты обоих операндов (которые в данном случае совпадают) и устанавливает биты результата по следующему правилу: каждый бит результата устанавливается в 1, если соответствующие биты операндов различаются, и сбрасывается в 0, если соответствующие биты операндов совпадают (рис. 17.1).

В предложении 17 в команде xor оба операнда совпадают. Легко сообразить, что в этом случае независимо от значения операнда после выполнения команды xor он станет равен нулю. Это дает основание использовать команду xor для обнуления операнда (вместо того, чтобы засылать в регистр 0 командой mov).

Первый операнд	0101 1111 0000 1111
Второй операнд	0110 1111 1111 0000
Результат (замещает первый операнд)	0011 0000 1111 1111

Рис. 17.1. Результат действия команды xor

Команда xor *reg,reg* занимает в памяти меньше места, чем команда mov *reg*,0, что и объясняет использование этого не очень наглядного приема.

Обнулив старший байт регистра AX и имея в его младшем байте введенную десятичную цифру в виде двоичного числа, мы на время сохраняем содержимое AX в регистре CX (предложение 18). Далее результат преобразования предыдущих цифр (или 0, если вводится первая цифра) переносится из DI в AX и умножается на 10 (предложения 19...21). Команда умножения mul предполагает наличие одного из сомножителей в AX. В качестве другого сомножителя не может выступать число, поэтому сомножитель 10 мы занесли в регистр BX. Результат умножения процессор заносит в два регистра: в DX (старшую половину результата) и в AX (младшую половину). В нашем случае результат умножения не может превысить 64 К, поэтому содержимым DX мы не интересуемся.

Получив в AX результат преобразования предыдущих цифр, умноженный на 10, мы прибавляем к нему текущую цифру из регистра CX, сохраняем в регистре DI и переходим на метку inpt с целью ввода следующей цифры.

При обнаружении кода клавиши Enter выполняется переход на метку done, полученное число переносится из DI в AX и процесс ввода и преобразования на этом завершается.

Для того чтобы проконтролировать работу программы, мы можем воспользоваться разработанными ранее подпрограммами bin_asc и wrd_asc, которые позволяют преобразовать содержимое регистра или ячейки памяти в 16-ричное число в символьном представлении. Для хранения этой символьной строки в программе предусмотрено поле result, начинающееся с символов '='. Эти символы введены для отделения на экране отображения результата контроля от результатов ввода числа. Учитывая, что символы самого числа должны начинаться с байта result+3, именно такое смещение загружено в регистр SI перед вызовом подпрограммы wrd_asc (предложение 26).

Разумеется, обе подпрограммы bin_asc и wrd_asc должны быть включены в состав исходного текста программы, как это и обозначено в примере 17.1.

Одним из недостатков программы является отсутствие анализа числа введенных символов. Между тем в машинное слово нельзя записать число, превышающее 65 535, поэтому ввод шестой и последующих десятичных цифр не имеет смысла. Для того чтобы ограничить число вводимых законных десятичных цифр и в то же время не учитывать случайно нажатые алфавитные и прочие клавиши, счетчик проходов программы следует установить в той точке, где начинается обработка законной десятичной цифры, т. е. между предложениями 12 и 13. Счетчик можно организовать следующим образом.

В начале программы надо инициализировать счетчик проходов, в качестве которого удобно использовать какой-либо свободный регистр, например SI:

```
mov    SI,5
```

а после предложения 12 включить в программу строки изменения и анализа содержимого счетчика:

```
        dec    SI
        jl     inpt
```

Команда jl не срабатывает, пока в SI положительное число или 0. Однако как только после пяти проходов (т. е. ввода пяти законных десятичных цифр) число в SI станет меньше 0, команда jl будет после ввода любого символа передавать управление назад на метку inpt, блокируя тем самым вывод символов на экран и их преобразование в числа. В этой ситуации программа будет ожидать нажатия клавиши Enter, чтобы перейти на строки завершения.

Еще один очевидный недостаток нашей простой программы – невозможность стереть (клавишей возврата на шаг) ошибочно введенные десятичные цифры. Однако включение в программу такой возможности привело бы к ее чрезмерному усложнению.

Создав и отладив пример 17.1, вы получите весьма полезный специализированный калькулятор: он переводит вводимые с клавиатуры десятичные числа в 16-ричные.

Статья 18. **Ввод с клавиатуры 16-ричных чисел**

Ввод с клавиатуры десятичных чисел является типичной операцией при решении математических или технических задач. Программисту, особенно любознательному, значительно чаще приходится иметь дело с 16-ричным представлением чисел. Как и в случае десятичных чисел, при вводе с клавиатуры 16-ричного числа в программу поступают коды ASCII его знаков (символов от '0' до '9' и от 'A' до 'F'). Программа ввода должна преобразовывать символы знаков в числовую форму и объединять эти числа в одном слове с учетом весов отдельных разрядов. Кроме того, необходимо предусмотреть анализ поступающих символы и отбраковку ошибочно введенных.

Рассмотрим программу, которая вводит с клавиатуры последовательность 16-ричных чисел длиной от 1 до 4 разрядов каждое и выводит их файл на диске. Такая программа может оказаться полезной для создания тестовых числовых файлов, используемых при отладке алгоритмов обработки числовой информации.

Пример 18.1 Ввод с клавиатуры последовательности 16-ричных чисел с выводом их в дисковый файл

```
;В сегменте команд
;Подпрограмма ввода чисел
hexin   proc
        mov    BX,0        ;Очистим вспомогательный регистр
inpt:   mov    AH,08h      ;Функция ввода символа
        int    21h
        cmp    AL,13       ;Клавиша Enter?
        je     done        ;Да, на вывод пробела
        cmp    AL,27       ;Клавиша Esc?
        je     done1       ;Да, на выход из подпрограммы
        cmp    AL,'0'      ;Меньше '0'?
        jb     inpt        ;Да, на повторение ввода
        cmp    AL,'9'      ;Не больше '9'?
        jbe    ok          ;Да, введена цифра
        cmp    AL,'F'      ;Больше 'F'?
        ja     inpt        ;Да, на повторение ввода
        cmp    AL,'A'      ;Меньше 'A'?
        jb     inpt        ;Да, на повторение ввода
ok:     mov    AH,02h      ;Введено разумное,
        mov    DL,AL       ;Выведем символ на экран
```

```
        int    21h
        cmp    AL,'9'         ;Введена цифра?
        ja     letter         ;Буква!
        sub    AL,'0'         ;Введена цифра, преобразуем в число
        and    AX,0Fh         ;Замаскируем остальные биты
        jmp    addd           ;На продолжение
letter: sub    AL,55          ;Введена буква, преобразуем в число
        and    AX,0Fh         ;Замаскируем остальные биты
addd:   mov    CL,4           ;Умножим предыдущий результат
        sal    BX,CL          ;на 16 сдвигом на 4 бита влево
        or     BX,AX          ;Добавим новую цифру
        jmp    inpt           ;Будем вводить дальше
done:   mov    AH,02h         ;Выведем пробел
        mov    DL,' '         ;между вводимыми числами
        int    21h
done1:  ret                   ;В BX введенное число
hexin   endp
;Фрагмент главной процедуры
;Создадим файл на диске
        mov    AH,3Ch         ;Функция создания файла
        mov    CX,0           ;Без атрибутов
        mov    DX,offset fname;Адрес имени файла
        int    21h
        mov    handle,AX      ;Сохраним дескриптор
;Вызов в цикле процедуры ввода 16-ричного числа
write:  call   hexin          ;Ввод числа
        cmp    AL,27          ;Вернулся код Esc?
        je     close          ;Да, на завершение
        mov    buf,BX         ;Перенесем результат в память
        mov    AH,40h         ;Функция записи в файл
        mov    BX,handle      ;Дескриптор файла
        mov    CX,2           ;Запись 2 байт
        mov    DX,offset buf;Адрес буфера вывода
        int    21h
        jmp    write          ;На ввод следующего числа
close:  mov    AH,09h         ;Выведем завершающее сообщение
        mov    DX,offset mesg
        int    21h
;В сегменте данных
fname   db 'nums.dat',0      ;Имя файла
handle  dw 0                  ;Ячейка для дескриптора
buf     dw 0                  ;Буфер вывода
mesg    db 'Формирование файла завершено$'
```

В отличие от примера 17.1, в котором ввод числа осуществлялся в основной программе, здесь ввод числа, представляющий собой относительно сложную операцию, вынесен в подпрограмму-процедуру hexin, которая многократно вызывается из основной программы. Ввод каждого очередного числа завершается нажатием клавиши Enter. Для завершения работы программы следует нажать клавишу Esc.

В подпрограмме hexin прежде всего очищается регистр BX, который будет использован для передачи в основную программу результирующего числа. Собственно ввод символов с клавиатуры осуществляется функцией DOS 08h, которая, как уже отмечалось, работает без эха (мы будем отображать на экране только правильно введенные знаки). Введенный символ проходит процедуру последовательного анализа. Если нажата клавиша Enter (ввод очередного числа закончен), на экран выводится пробел и процедура hexin завершается. Если нажата клавиша Esc (команда на завершение программы), процедура hexin завершается с обходом строк вывода пробела.

Далее происходит отбор знаков, представляющих собой законные 16-ричные цифры. Если введенный символ принадлежит диапазонам '0'...'9' или 'A'...'F', он вводится функцией 02h на экран и выполняется его преобразование в число. В случае ввода ошибочного символа происходит возврат на повторный ввод.

Преобразование в число заключается в вычитании из кода ASCII введенного символа кода символа '0' для цифр и числа 55 для букв (поскольку 16-ричный символ 'A' с кодом 41h=65 соответствует числу 10). Полученное число занимает 4 младших бита в регистре AL; остальные разряды регистра AX очищаются командой and. Результат предыдущих преобразований в регистре BX умножается на 16 сдвигом влево на 4 разряда, к нему командой or добавляется очередной 16-ричный разряд, после чего происходит переход на метку inpt для ввода следующей цифры.

Основная программа с помощью функции DOS 3Ch создает на диске файл с указанным именем, сохраняет дескриптор этого файла и входит в цикл вызова процедуры hexin. После каждого вызова hexin анализируется содержимое регистра AL, куда в подпрограмме поступает код введенного символа, и при обнаружении в нем числа 27 (код ASCII клавиши Esc) осуществляется выход из цикла, вывод предупреждающего сообщения и завершение программы.

Если код Esc не обнаружен, полученное из hexin число переносится в ячейку памяти buf и вызывается функция DOS 40h записи в файл 2 байтов из этой ячейки. Вывести число в файл непосредственно из регистра BX нельзя, так как функция 40h умеет выводить информацию в файл только из памяти, адрес которой указывается в регистре DX.

Существенным недостатком нашей программы является необходимость при вводе 16-ричного числа пользоваться только прописными буквами. Строчные буквы программа отбраковывает как ошибочные. По-настоящему следовало бы после каждого ввода строчной буквы преобразовывать ее в прописную вычитанием из ее кода 20h. Это, однако, усложнило бы алгоритм анализа введенного кода.

Статья 19. **Двоично-десятичные числа**

В ряде прикладных областей для представления чисел используется специальный формат, называемый двоично-десятичным (binary-coded decimal, BCD). В таком формате выдают данные некоторые измерительные приборы; он же используется КМОП-микросхемой компьютера для хранения информации о текущем времени. В процессоре предусмотрены команды для обработки таких чисел.

Двоично-десятичный формат существует в двух разновидностях: упакованный и распакованный. В первом случае в байте записывается двухразрядное десятичное число от 00 до 99. Каждая цифра числа занимает половину байта и хранится в двоичной форме.

Из рис. 19.1, где приведен пример кодирования упакованного двухразрядного двоично-десятичного числа, можно заметить, что для записи в байт десятичного числа в двоично-десятичном формате достаточно сопроводить записываемое десятичное число символом h.

```
1001 0111  Двоичное содержимое байта
 9    7    Десятичное обозначение числа
 9    7h   16-ричное обозначение числа
```

Рис. 19.1. Упакованный двоично-десятичный формат

В машинном слове или в 16-разрядном регистре можно хранить в двоично-десятичном формате 4-разрядные десятичные числа от 0000 до 9999 (рис. 19.2).

1000 0110 0000 1001	Двоичное содержимое слова	
8 6 0 9	Десятичное обозначение числа	
8 6 0 9h	16-ричное обозначение числа	

Рис. 19.2. Слово, содержащее 4-разрядное десятичное число в упакованном BCD-формате

Распакованный формат отличается от упакованного тем, что в каждом байте записывается лишь одна десятичная цифра (по-прежнему в двоичной форме). В этом случае в слове можно записать десятичные числа от 00 до 99 (рис. 19.3)

0000 1001 0000 1000	Двоичное содержимое слова
9 8	Десятичное обозначение числа
0 9 0 8h	16-ричное обозначение числа

Рис. 19.3. Запись десятичного числа в распакованном виде

При хранении десятичных чисел в аппаратуре обычно используется более экономный упакованный формат; умножение и деление выполняются только с распакованными числами, операции же сложения и вычитания применимы и к тем и к другим.

Разработаем процедуры для выполнения арифметических операций с информацией о текущем времени, получаемой из часов реального времени компьютера.

Современные компьютеры оснащаются КМОП-микросхемой с батарейным питанием, которая служит, во-первых, для хранения информации о конфигурации компьютера (количество и типы дисков, объем памяти и пр.), а во-вторых, для отсчета реального календарного времени. Поскольку эта микросхема питается от встроенной подзаряжаемой батарейки, часы реального времени продолжают работать, даже если компьютер выключен. В процессе начальной загрузки программы DOS считывают показания часов реального времени, сохраняют их в ячейках системной области и модифицируют в дальнейшем эти ячейки в соответствии с ходом работы системного таймера.

Для чтения и изменения текущего времени и даты используются функции DOS 2Ah (получить дату), 2Bh (установить дату), 2Ch (получить время) и 2Dh (установить время), при этом DOS не только изменяет время текущего сеанса, но и модифицирует установки КМОП-часов реального времени. Однако к часам реального времени можно обратиться и в обход DOS, с помощью прерывания BIOS 1Ah. Оно предоставляет больше возможностей, чем функции DOS, в частности позволяет установить "будильник", который в заданный момент времени даст сигнал прерывания. Этот сигнал с помощью обработчика прерывания можно обнаружить и отработать требуемым образом.

При обращении к системному таймеру в помощью функций DOS время и дата задаются и получаются в виде обычных двоичных чисел. Однако в КМОП-микросхеме время и дата хранятся в упакованном двоично-десятичном формате по две десятичные цифры в одном байте. В таком же формате эти данные принимаются и возвращаются функциями прерывания 1Ah BIOS. Таким образом, при использовании для операций с текущим временем прерывания BIOS (а такжс при программировании КМОП-микросхемы непосредственно через ее порты) необходимо обрабатывать числа в формате BCD.

И при получении, и при установкс времени с помощью прерывания 1Ah число часов находится в регистре CH, число минут – в CL и число секунд – в DH, т. е. для хранения полного значения времени требуется 3 байта. Байты секунд и минут могут содержать числа от 00 до 59; байт часов – от 00 до 23.

Получив с помощью функции BIOS три составляющих времени из КМОП-микросхемы, преобразовав эти числа в символьную форму и выведя полученную строку на экран, мы получим программу-часы (пример 19.1), которая при ее запуске будет выводить на экран текущее реальное время.

Поскольку значение времени включает три двухразрядных BCD-числа, процедуру преобразования придется применить трижды, и ее полезно оформить в виде подпрограммы, которая в примере 19.1 названа bcd_asc.

Пример 19.1. Преобразование упакованных двоично-десятичных чисел в символьную форму

```
.386                            ;Будут использоваться команды МП 80386
        mov     AH,02h          ;Функция чтения времени
        int     1Ah             ;Прерывание BIOS
        mov     AL,CH           ;Заберем в AL часы
        call    bcd_asc         ;Преобразуем в символы
        mov     word ptr time,AX;Занесем в строку
        mov     AL,CL           ;Заберем в AL минуты
        call    bcd_asc         ;Преобразуем в символы
        mov     word ptr time+3,AX;Занесем в строку
        mov     AL,DH           ;Заберем в AL секунды
        call    bcd_asc         ;Преобразуем в символы
        mov     word ptr time+6,AX;Занесем в строку
        mov     AH,09h          ;Функция вывода на экран
        mov     DX,offset clock;Смещение строки
        int     21h             ;Вызов DOS
;Подпрограмма преобразования упакованного BCD-числа в символы
;На входе BCD в AL. На выходе символы в AX
bcd_asc proc
        mov     AH,AL           ;(1)Занесем число и в AH
        and     AH,0Fh          ;(2)Выделим в AH младшую цифру
        add     AH,'0'          ;(3)Преобразуем в ASCII
        and     AL,0F0h         ;(4)Выделим в AL старшую цифру
        shr     AL,4            ;(5)Сдвиг в начало регистра. Команда 80386!
        add     AL,'0'          ;(6)Преобразуем в ASCII
        ret                     ;(7)
bcd_asc endp
;В полях данных
clock   db 'Текущее время '
time    db 0,0':',0,0,':',0,0,'$'
```

В главной процедуре после получения текущего времени с помощью функции 02h прерывания 1Ah составляющие времени вызовом процедуры bcd_asc последовательно преобразуются в символьную форму и помещаются на предназначенные для них места в строке clock, которая содержит поясняющий текст ('Текущее время') и разделители (в виде двоеточий) между пустыми пока полями для компонентов времени. Назначение процедуры bcd_asc – преобразование упакованного двухразрядного двоично-десятичного числа в символьную форму (т. е. в два кода ASCII соответствующих символов).

Исходное двухразрядное число, находящееся в AL, копируется в AH. В этом регистре командой and выделяется младший десятичный разряд числа (предложение 2), который затем прибавлением к нему кода символа '0' преобразуется в код ASCII (предложение 3). В регистре AL аналогичная операция выполняется со старшим десятичным разрядом. Он выделяется командой and (предложение 4), сдвигается в начало регистра командой shr (предложение 5), после чего преобразуется в код ASCII. Резуль-

тат преобразования в виде двух кодов ASCII остается в регистре AX, откуда в главной процедуре он может быть перенесен в выводимую на экран строку.

Следует обратить внимание на порядок формирования десятичных разрядов исходного числа в регистре AX. В старшем байте этого регистра (AH) мы формируем код ASCII *младшего* разряда числа, а в младшем байте (AL) – код *старшего* разряда. Обратный порядок разрядов связан с тем, что мы привыкли изображать числа начиная со старших разрядов и, соответственно, в символьной строке, которая будет выводиться на экран, в младших (левых) байтах должны размещаться старшие разряды отображаемых чисел, а в старших (правых) байтах – младшие. Такой же обратный порядок десятичных разрядов должен быть и в регистре, из которого 2-байтовые элементы строки переносятся в память.

Статья 20. **Деассемблирование и машинные коды команд**

Как видно из примеров, приведенных в этой книге, программы на языке ассемблера пишутся с использованием достаточно наглядных мнемонических обозначений команд и способов адресации. В процессе составления программы программисту, как правило, не приходится сталкиваться с машинными кодами команд. Однако многие аспекты деятельности программиста требуют хотя бы поверхностного знакомства с правилами образования машинных кодов команд. К таким аспектам можно отнести: отладку и исследование работы сложных программ; оптимизацию программ по памяти или времени выполнения; расшифровку и изучение загрузочных модулей системного и прикладного программного обеспечения. Последнее является не только весьма полезной для самообразования, но часто и необходимой операцией.

В настоящее время все программное обеспечение персональных компьютеров поступает на рынок в виде загрузочных модулей, готовых к выполнению. Исходные тексты программ практически всегда отсутствуют. При возникновении необходимости внесения в программы хотя бы незначительных изменений приходится тем или иным способом "деассемблировать" загрузочные модули, т. е. преобразовывать их в текст на языке ассемблера, отыскивать интересующие пользователя участки и вносить затем изменения непосредственно в машинные коды команд или данных.

Для деассемблирования можно воспользоваться специальными инструментальными программами – деассемблерами, например программой Sourcer фирмы V Communications. При этом, однако, следует иметь в виду, что деассемблер всегда выполняет свою работу небезошибочно. Ведь команды в принципе неотличимы от данных, в результате чего деассемблер обычно часть кодов команд расшифровывает как данные и, наоборот, часть данных представляет в виде машинных команд. Для того чтобы разобраться в получившейся путанице, необходимо иметь представление о правилах кодирования команд процессора. С другой стороны, во многих случаях изменения, вносимые в загрузочный модуль, столь незначительны, что не имеет смысла расшифровывать всю программу. Достаточно найти в ней несколько требуемых байтов. Эту работу вполне можно выполнить "вручную", с помощью какой-либо программы, выводящей на экран 16-ричное содержимое файлов, например PC Tools или Norton

Commander. В качестве примера "подправления" программных продуктов можно привести используемую иногда процедуру изменения программных строк анализа версии DOS с целью эксплуатации старой программы в новой версии DOS (естественно, этот метод применим лишь в тех случаях, когда анализ версии DOS носит формальный характер, для самой же программы по существу версия DOS безразлична).

Система кодирования команд в процессорах Intel весьма сложна и иногда неоднозначна. С другой стороны, она обеспечивает достаточно гибкие механизмы адресации и эффективное использование памяти, отводимой программе. Рассмотрим общие закономерности образования машинных кодов команд процессора 8086. Приводимый материал в целом справедлив и для 32-разрядных процессоров, однако включение в архитектуру этих процессоров дополнительных способов адресации, а также возможность работы с 32-битовыми операндами еще более усложняет систему кодирования команд.

В общем виде структуру машинной команды можно представить следующим образом:

Префикс – Код операции – Адресация – Смещение – Операнд

Каждый элемент команды занимает один или несколько байтов. Необходимым является только код операции (коп), который характеризует выполняемую операцию (пересылка, сложение и т. д.) и должен присутствовать в любой команде.

Префикс команды занимает 1 байт и задает либо замену сегментного регистра, используемого по умолчанию (ES:, CS:), либо повторение команды CX раз (rep, repe).

Формат префикса замены сегмента приведен на рис. 20.1; в табл. 20.1 указаны значения поля SR (код сегментного регистра) для этого байта.

Биты	7	6	5	4	3	2	1	0
	0	0	1	SR		1	1	0

Рис. 20.1. Структура байта префикса замены сегмента

Таблица 20.1. Расшифровка поля SR

Поле SR	*Регистр*
00	ES
01	CS
10	SS
11	DS

Префиксы повторения rep и repe имеют коды F3h, а префикс repne – код F2h (префиксы rep и repe используются в разных командах, поэтому не возникает неоднозначности при расшифровке кода).

Код операции занимает обычно 1 байт и характеризует команду, при этом многим командам присущи несколько кодов операций, которые частично определяют использованный способ адресации.

Байт адресации в оригинальной литературе носит название MOD R/M и состоит из трех полей: Mod, Reg (или Reg/коп) и R/M (рис. 20.2).

Биты	7	6	5	4	3	2	1	0
	Mod		Reg/коп			R/M		

0 0 В коде команды отсутствует смещение (при адресации к памяти)
0 1 Смещение в коде команды есть и занимает 1 байт
1 0 Смещение в коде команды есть и занимает 1 слово
1 1 Операнды являются регистрами

Рис. 20.2. Структура байта MOD R/M с расшифровкой поля Mod

Как видно из рис. 20.2, поле Mod занимает 2 бита; в нем указывается, выполняется ли адресация к памяти или к регистрам. Если оба операнда являются регистрами, Mod=11; при других значениях Mod один из операндов находится в памяти. Как известно (см. статью 10), для обращения к памяти предусмотрено несколько способов адресации: прямая, косвенная регистровая и косвенная регистровая со смещением. Если Mod=00, смещение в команде отсутствует и, следовательно, выполняется команда типа inc word ptr [BX] или mov AX,[BX][SI]. Если Mod=01, в команде указано смещение, причем оно лежит в диапазоне знаковых байтовых чисел: mov [BX][SI+127], CX или inc word ptr [BX-2]. Если Mod=10, в команде присутствует смещение длиной 2 байта: dec mem[SI] или neg byte ptr [BX][DI+128] (адрес ячейки памяти mem с очевидностью является словом, а число 128 выходит за рамки байтовых знаковых чисел).

Поле Reg/коп для некоторых команд используется как расширение байта кода операции; в других случаях в нем указывается условный код одного из адресуемых регистров (приемника или источника), причем по по младшему биту кода операции процессор определяет, надо ли использовать полный 16-разрядный регистр или одну из его половин. В табл. 20.2 показано соответствие кодов поля Reg регистрам процессора.

Таблица 20.2. Коды регистров поля Reg

Поле Reg	*Регистр 16 бит*	*Регистр 8 бит*
000	AX	AL
001	CX	CL
010	DX	DL
011	BP	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

Поле R/M (Register/Memory, регистр/память) используется для идентификации способа адресации. Расшифровка кода в этом поле зависит от значения поля Mod того же байта. Если Mod=11, в команде используется регистровая адресация и в поле R/M указывается код регистра (который может быть как источником, так и приемником); при других значениях Mod выполняется обращение к памяти и в поле R/M закодирована комбинация регистров, используемых для косвенного обращения к памяти, а также наличие или отсутствие в команде смещения. В табл. 20.3 приведена расшифровка кодов в поле R/M для различных значений Mod.

Обратите внимание на отсутствие во второй графе таблицы кода R/M для описания адресации через BP. Это не означает, что регистр BP нельзя использовать для косвенного обращения к памяти. Команда типа mov [BP],CX вполне законна, но трансля-

тор преобразует ее в вид mov [BP+*смещение*],CX, где смещение указывается в третьем байте команды и равно нулю.

Таблица 20.3. Расшифровка поля R/M

R/M	*Mod=00*	*Mod=01 или 10*	*Mod=11*	
000	[BX][SI]	[BX][SI+смещение]	AX	AL
001	[BX][DI]	[BX][DI+смещение]	CX	CL
010	[BP][SI]	[BP][SI+смещение]	DX	DL
011	[BP][DI]	[BP][DI+смещение]	BX	BL
100	[SI]	[SI+смещение]	SP	AH
101	[DI]	[DI+смещение]	BP	CH
110	Прямая	[BP+смещение]	SI	DH
111	[BX]	[BX+смещение]	DI	BH

Некоторые однооперандные команды в случае регистрового способа адресации (push BX, inc CX) кодируются одним байтом, в который входит, наряду с кодом операции, и код регистра-операнда. Формат таких команд приведен на рис. 20.3; коды регистров в этом случае соответствуют табл. 20.2 (столбец 16-разрядных регистров).

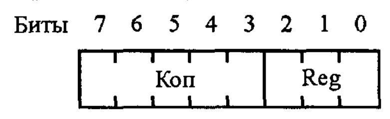


Рис. 20.3. Структура байта 1-байтовых команд

Аналогичная ситуация возникает при использовании в некоторых командах в качестве операнда сегментного регистра. Такие команды могут иметь для этих случаев особый код операции, в котором предусмотрено 2-битовое поле SR для обозначения сегментного регистра. Коды регистров соответствуют табл. 20.1.

В первом байте команды – байте кода операции, в ряде случаев целесообразно выделять два младших бита. Формат такого байта изображен на рис. 20.4.

Бит 0 (W) принимает значение 1, если команда оперирует со словом. В случае байтовой операции W=0.

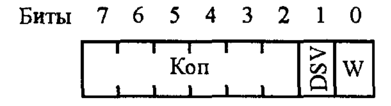


Рис. 20.4. Структура байта кода операции для некоторых команд

Бит 1 в одних командах обозначает направление операции (D), в других – задает характеристики непосредственного операнда (S). В командах сдвигов этот бит (тогда он обозначается V) определяет наличие счетчика сдвигов.

Если D=1, операция выполняется в регистр Reg; если D=0 – из регистра Reg.

Если SW=01, то выполняется операция со словом (W=1), причем операнд образует 16 бит непосредственных данных (add mem,1234h). Если SW=0, то выполняется операция со словом, но в качестве операнда в команде указано байтовое знаковое число (add mem,5), и этот байт непосредственных данных расширяется со знаком для образования "законного" 16-битового операнда.

На рис. 20.5...20.7 приведены примеры кодирования команд.

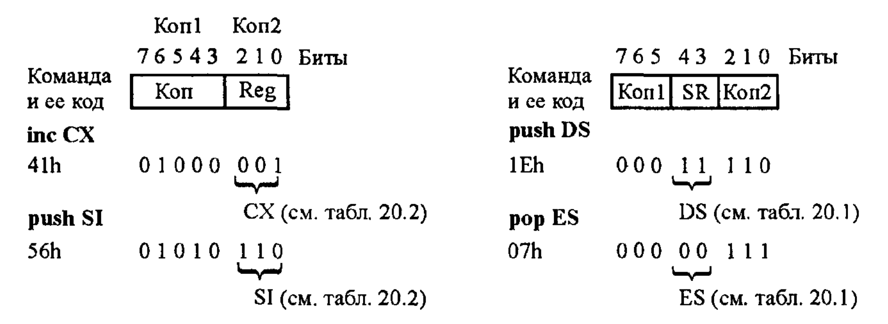


Рис. 20.5. Примеры 1-байтовых команд с обращением к регистрам общего назначения и сегментным

Команда и ее код | 7 6 5 4 3 2 1 0 | 7 6 5 4 3 2 1 0 | Смещение

Коп DW | Mod Reg R/M | Младший байт | Старший байт

add DI, CX

03h F9h 000000 11 11 111 001 — —

Регистр; DI; CX } См. табл. 20.3

16 бит

add [DI], CX

01h 0dh 000000 01 00 001 101 — —

Без смещения; CX; [DI] } См. табл. 20.3

16 бит

Из регистра (CX)

add [DI+5], CX

01h 4Dh 05h 000000 01 01 001 101 05h —

Смещение 8 бит; CX; [DI] } См. табл. 20.3

16 бит

Из регистра (CX)

mov memb [BX] [DI], DH

88h B1h 2235h 100010 00 10 110 001 35h 22h

(memb – ячейка памяти, объявленная как байт)

Смещение 16 бнт; DH; [BX][DI+смещение] } См. табл. 20.3

8 бит

Из регистра (DH)

Рис. 20.6. Примеры команд, использующих смещение

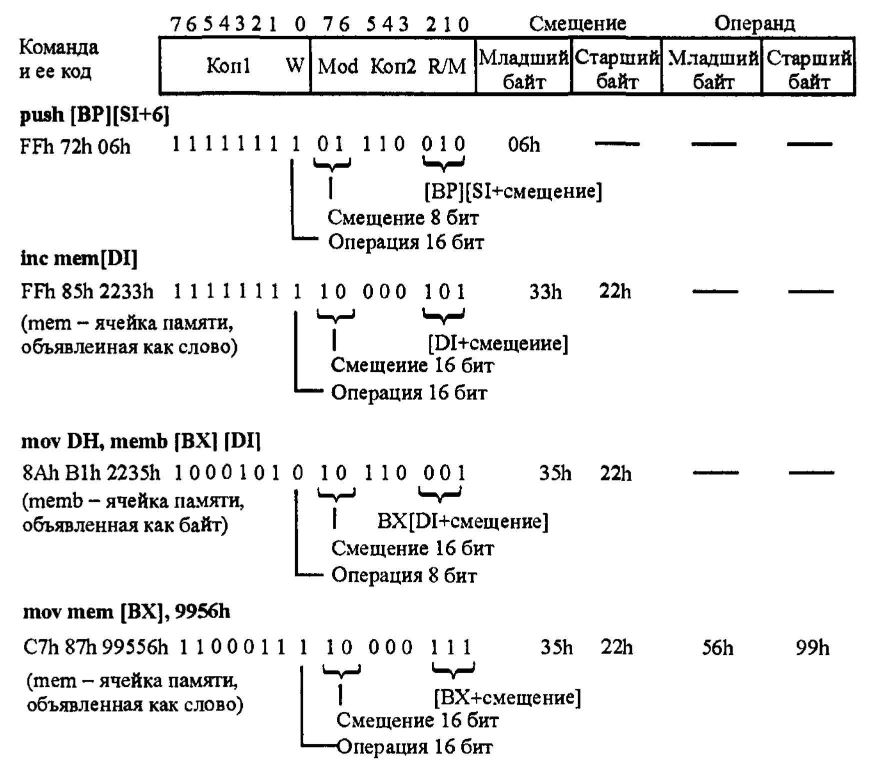


Рис. 20.7. Примеры команд с расширенным кодом операции

Статья 21. **Макрокоманды**

Программы, написанные на языке ассемблера, часто содержат повторяющиеся участки текста с одинаковой структурой. Такие участки текста можно оформить в виде так называемых макроопределений (макрокоманд, макросов), характеризующихся произвольными именами и списками формальных параметров. После того как такое макроопределение сделано, появление в программе предложения, содержащего имя макроопределения и список фактических параметров, приводит к генерации всего требуемого текста (макрорасширения), в котором все формальные параметры заменяются фактическими. Варьируя фактические параметры, можно, сохраняя неизменной структуру макрорасширения, изменять отдельные его элементы.

Макроопределение должно начинаться заголовком с именем макроопределения и директивой macro, за которой может следовать список формальных параметров (в простейшем случае параметров может и не быть). Вслед за заголовком располагается текст макроопределения. Заканчивается макроопределение директивой endm.

Пусть в программе требуется неоднократно сохранять в стеке содержимое трех регистров, но в каждом конкретном случае имена регистров и их порядок отличаются. Оформим эти действия в виде макроопределения:

```
push3   macro a,b,c
        push  a
```

```
        push    b
        push    c
        endm
```

Появление в исходном тексте программы предложения

```
        push3  AX,BX,CX
```

приведет к генерации следующего фрагмента:

```
        push    AX
        push    BX
        push    CX
```

Если же в исходном тексте имеется строка

```
        push3  DX,ES,BP
```

то соответствующее макрорасширение будет иметь вид:

```
        push    DX
        push    ES
        push    BP
```

В качестве фактических параметров могут выступать любые обозначения ассемблера, допустимые для данной команды. В частности, макровызов

```
        push3  mem,[BX],ES:[17h]
```

приведет к следующему макрорасширению:

```
        push    mem
        push    [BX]
        push    ES:[17h]
```

Если какие-то строки макроопределения должны быть помечены (с целью организации переходов или циклов), то обозначения меток должны быть объявлены локальными с помощью оператора local. В этом случае ассемблер, генерируя макрорасширения, будет создавать собственные обозначения меток, не повторяющиеся при повторных вызовах одной и той же макрокоманды:

```
stars   macro
local   outpt
        mov     CX,10           ;Счетчик цикла
        mov     AH,02h          ;Функция вывода символа
        mov     DL,'*'          ;Символ
outpt:  int     21h             ;Вызов DOS
        loop    outpt           ;Цикл из CX шагов
        endm
```

Макрос stars выводит на экран строку из 10 звездочек, которая может служить разделителем фрагментов текста в экранном кадре. Если в текст программы включить два макровызова stars:

```
...
stars
...
stars
...
```

то их макрорасширения, подставляемые ассемблером в текст программы, будут выглядеть следующим образом:

```
        mov     CX,10
        mov     AH,02h
        mov     DL,'*'
??0000: int     21h
```

```
        loop   ??0000
...
        mov    CX,10
        mov    AH,02h
        mov    DL,'*'
??0001: int    21h
        loop   ??0001
```

При повторных подстановках макроопределения ассемблер заменяет обозначение метки outpt на различающиеся обозначения ??0000, ??0001 и т. д., обеспечивая тем самым правильное выполнение команд циклов и переходов.

Макросы могут быть вложенными. Предыдущий пример, очевидно, плох тем, что звездочки выводятся не на новой строке, а начиная от текущего положения курсора. Макрос stars будет иметь больше практического смысла, если включить в него до и после вывода звездочек перевод курсора в начало следующей строки экрана. Эти действия можно оформить в виде отдельного макроса:

```
crlf    macro
        mov    AH,02h        ;Функция вывода символа
        mov    DL,13         ;Код возврата каретки
        int    21h           ;Вызов DOS
        mov    DL,10         ;Код перевода строки
        int    21h           ;Вызов DOS
        endm
```

Макроопределение stars теперь можно модифицировать следующим образом:

```
stars   macro
local   outpt
        crlf                 ;Вложенный макровызов
        mov    CX,10         ;Счетчик цикла
        mov    AH,02h        ;Функция вывода символа
        mov    DL,'*'        ;Символ
outpt:  int    21h           ;Вызов DOS
        loop   outpt         ;Цикл из CX шагов
        crlf                 ;Вложенный макровызов
        endm
```

Если макроопределения носят частный характер и предназначены для использования лишь в конкретной программе, то их можно включить непосредственно в исходный текст программы, как это сделано в примере 21.1.

Пример 21.1. Использование макрокоманд

```
;Опишем макроопределения
crlf    macro
        ...    ;Текст макроопределения crlf
        endm
stars   macro
        ...    ;Текст макроопределения stars
        endm
;Текст программы с макровызовами
text    segment
        assume cs:text,ds:data
begin:  mov    AX,data
        mov    DS,AX
        stars                ;1-й макровызов
        mov    AH,09h
        mov    DX,offset mesg1
        int    21h
        stars                ;2-й макровызов
```

```
        mov     AH,09h
        mov     DX,offset mesg2
        int     21h
        mov     AX,4C00h
        int     21h
text    ends
data    segment
mesg1   db 'Первый фрагмент текста$'
mesg2   db 'Второй фрагмент текста$'
data    ends
stk     segment stack
        db 256 dup (0)
stk     ends
        end     begin
```

Обычно у программиста в процессе работы накапливаются полезные макроопределения общего назначения: фрагменты программной задержки, останова программы до нажатия клавиши, преобразования двоичных чисел в символьную форму и т. д. Такие макроопределения целесообразно поместить в макробиблиотеку.

Макробиблиотека представляет собой файл с текстами макроопределений, которые записываются в этот файл точно в таком же виде, как и в текст программы. Файл макробиблиотеки может иметь любое имя и расширение, например: MYMACRO.MAC. В программе же в этом случае остаются только макровызовы. Для того чтобы ассемблер мог подставить вместо макровызова соответствующие макрорасширения, в тексте программы следует объявить имя макробиблиотеки с помощью директивы include:

```
include mymacro.mac
```

После этого в программе можно использовать любые макрокоманды из этой макробиблиотеки.

Оформите приведенные выше макросы crlf и stars в виде файла макробиблиотеки. Уберите макроопределения из текста примера 21.1 и включите в программу директиву include. Убедитесь, что трансляция выполняется успешно и программа работает так же, как и раньше.

Раздел второй

АППАРАТНАЯ ОРГАНИЗАЦИЯ КОМПЬЮТЕРА

Статья 22. **Память**

Современные компьютеры оснащаются оперативной памятью объемом 32... ...64 Мбайт и более. Плюс к этому в состав компьютера входит несколько постоянных запоминающих устройств (ПЗУ) различной конструкции и назначения. Однако полностью наличная оперативная память может быть использована только в защищенном режиме. Этот режим устанавливается, в частности, если загружается одна из операционных систем защищенного режима, например Windows или OS/2. В реальном режиме под управлением операционной системы MS-DOS микропроцессор может обращаться к ячейкам памяти лишь в пределах первого мегабайта адресного пространства.

Не следует думать, что термины "адресное пространство" и "память" эквивалентны. Адресное пространство – это просто набор адресов, которые может формировать процессор; совсем не обязательно все эти адреса отвечают реально существующим ячейкам памяти. В зависимости от модификации компьютера и состава его периферийного оборудования распределение адресного пространства может несколько различаться. Тем не менее размещение основных компонентов системы довольно строго унифицировано. Типичная схема использования адресного пространства приведена на рис. 22.1.

Первые 640 Кбайт адресного пространства с адресами от 00000h до 9FFFFh (и соответственно, с сегментными адресами от 0000h до 9FFFh) отводятся под основную оперативную память, которую еще называют стандартной или обычной. В начало этой области при загрузке компьютера загружаются таблицы и программы DOS, которые могут занимать до нескольких десятков килобайт.

Начальный килобайт оперативной памяти занят векторами прерываний, которые обеспечивают работу системы прерываний компьютера. Всего здесь содержится 256 векторов по 4 байта каждый. Векторы заполняются автоматически в процессе начальной загрузки компьютера, однако при разработке пользователем прикладных обработчиков прерываний программисту приходится обращаться к этой области и заполнять отдельные векторы адресами прикладных обработчиков.

Вслед за векторами прерываний располагается так называемая область данных BIOS, которая занимает всего 256 байт, начиная с сегментного адреса 40h. Сама BIOS (Basic In-Out System, базовая система ввода-вывода) является частью операционной системы, хранящейся в ПЗУ, т. е. в таком устройстве, содержимое которого не стирается при выключении компьютера. Это запоминающее устройство (ПЗУ BIOS) располагается на системной плате компьютера. В функции BIOS входит тестирование узлов компьютера при каждом его включении, загрузка в оперативную память собственно операционной системы MS-DOS, хранящейся на магнитных дисках, а также управление штатной аппаратурой компьютера – клавиатурой, экраном, таймером и пр. Управляющие схемы ПЗУ BIOS настраиваются так, что ПЗУ оказывается отображенным на

адресное пространство в самом конце первого мегабайта, начиная с сегментного адреса F000h.

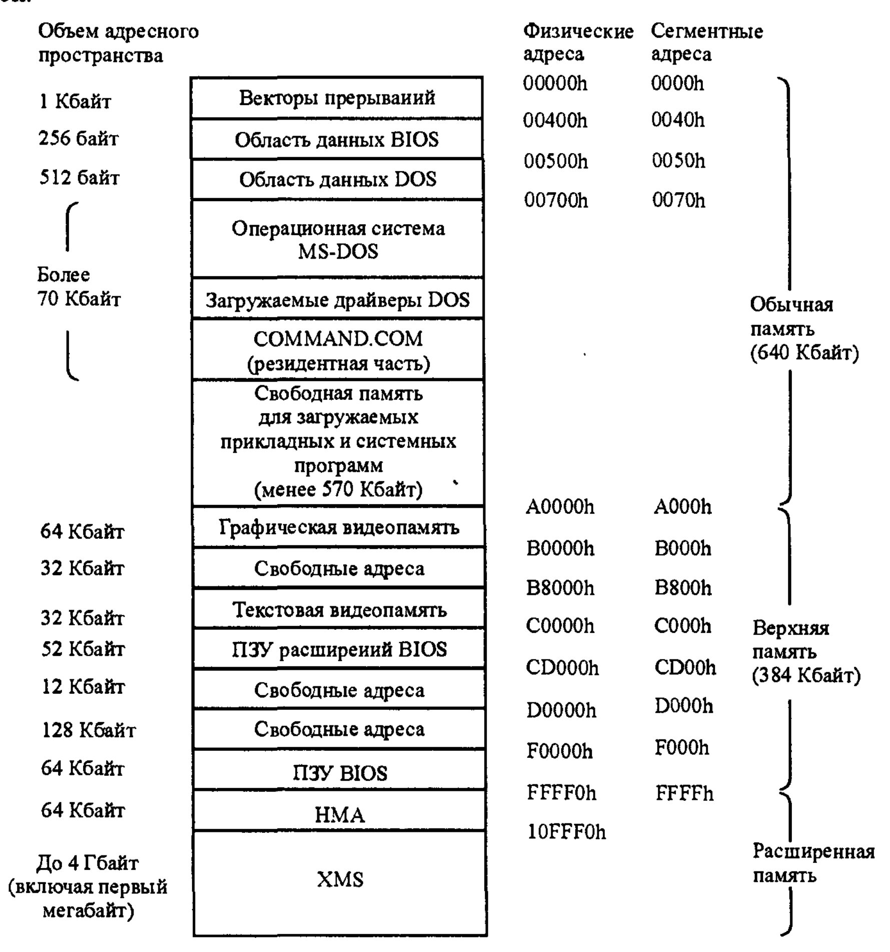


Рис. 22.1. Типичное распределение адресного пространства

В области данных BIOS, которая заполняется, как и область векторов, автоматически при включении компьютера, хранится разнообразная информация, используемые программами BIOS в своей работе. Так, здесь размещаются адреса видеоадаптера, а также последовательных и параллельных портов компьютера; данные, характеризующие текущее состояние видеосистемы (форма курсора и его положение на экране, видеорежим, текущая видеостраница и пр.); ячейки для отсчета текущего времени, модифицируемые системным таймером, работающим в режиме прерываний; буфер клавиатуры, куда поступают коды нажимаемых пользователем клавиш, и многое другое. Информация, хранящаяся в области данных BIOS, частично остается неизменной, а частично модифицируется программами BIOS по ходу работы компьютера (например, позиция курсора или содержимое буфера клавиатуры). Как правило, программист не обращается к ячейкам области данных BIOS напрямую, однако иметь представле-

ние о составе этой области и назначении ее ячеек во многих случаях оказывается необходимым. С некоторыми из этих ячеек мы столкнемся при рассмотрении примеров конкретных программ. В табл. 22.1 приведено назначение наиболее интересных для прикладного программиста ячеек области данных BIOS.

Таблица 22.1. Содержимое некоторых полей области данных BIOS (адреса указаны в виде смещений от начала области)

Адрес	*Размер, байт*	*Типичное значение*	*Назначение*
00h	2	03F8h	Базовый порт COM1
02h	2	02F8h	Базовый порт COM2
08h	2	0378h	Базовый порт LPT1
10h	2	C463h	Состав установленного оборудования
13h	2	0280h=640 Кбайт	Основная память, Кбайт
17h	1	00h	Первое слово флагов клавиатуры
18h	1	00h	Второе слово флагов клавиатуры
19h	1	00h	Буфер накопления при вводе Alt+цифры на дополнительной цифровой клавиатуре
1Ah	2	001Eh...003Ah	Головной указатель буфера клавиатуры
1Ch	2	001Eh...003Eh	Хвостовой указатель буфера клавиатуры
49h	1	03h	Текущий видеорежим
4Ah	2	50h=80	Ширина экрана (число знаков)
4Ch	2	1000h=4 Кбайт	Размер видеостраницы
4Eh	2	0000h	Смещение в видеопамяти текущей видеостраницы
50h	16	–	Позиции курсора на каждой видеостранице
60h	2	0607h	Форма курсора
62h	2	03D4h	Управляющий порт видеоконтроллера
67h	4	–	Адрес возврата после сброса процессора
6Ch	4	–	Счетчик прерываний системного таймера (18,2 Гц)
72h	2	0000h	Режим начальной загрузки. 0000h – полный цикл POST. 1234h – укороченный цикл POST (после Ctrl/Alt/del)
77h	1	01h	Число жестких дисков
80h	2	001Eh	Адрес начала буфера клавиатуры
82h	2	003Eh	Адрес за концом буфера клавиатуры
84h	1	18h=24	Число строк на экране – 1
85h	2	10h=16	Число строк развертки на символ
F0h	16	нули	Область межзадачных связей

В области памяти начиная с сегментного адреса 50h содержатся некоторые системные данные DOS. Эта область недокументирована и напрямую обращаться к ней не приходится. Вслед за областью данных DOS располагается собственно операционная

система MS-DOS, загружаемая из файлов IO.SYS и MSDOS.SYS. Система требует для своего размещения около 70 Кбайт.

Если в файл CONFIG.SYS включены директивы DEVICE= для загрузки устанавливаемых драйверов (SMARTDRV.SYS, EMM386.EXE, ANSI.SYS и др.), то они загружаются вслед за системой. Наконец, ниже драйверов размещается резидентная часть командного процессора COMMAND.COM, занимающая около 3 Кбайт. В функции резидентной части COMMAND.COM входит обработка команд оператора Ctrl+C и Ctrl+Break, а также критических ошибок, вывод сообщений об ошибках, завершение текущей программы, загрузка транзитной части COMMAND.COM. Транзитная, загружаемая часть COMMAND.COM размещается в самом конце оперативной памяти, затирается при загрузке больших программ и после завершения таких программ должна загружаться с диска заново.

Вся оставшаяся память до границы 640 Кбайт свободна для загрузки любых системных или прикладных программ. Как правило, в начале сеанса в память загружают резидентные программы (русификатор, антивирусные программы). При наличии резидентных программ объем свободной памяти уменьшается. Практически учитывая необходимость загрузки драйверов (например, компакт-дисков), а также резидентных программ (русификатора, поддержки компакт-дисков и мыши и др.), при таком конфигурировании компьютера объем свободной памяти вряд ли превысит 500...510 Кбайт.

Оставшиеся 384 Кбайт адресного пространства, называемые старшей или верхней (upper) памятью, первоначально были предназначены для размещения ПЗУ. Практически под ПЗУ занята только часть адресов. В самом конце адресного пространства, в области F0000h...FFFFFh, располагается основная часть ПЗУ BIOS, а начиная с адреса C0000h – так называемое ПЗУ расширений BIOS для обслуживания графических адаптеров и дисков. Часть адресов старшей памяти отводится для адресации видеопамяти графического адаптера. Физически видеопамять обычно находится на плате адаптера и является, таким образом, частью видеосистемы. Однако системы управления видеопамятью настраиваются так, что видеопамять оказывается отображенной на участки адресного пространства, указанные на рис. 22.1. Запись данных по этим адресам приводит к появлению на экране терминала изображений символов (или, в графическом режиме, отдельных точек).

В состав современных компьютеров наряду со стандартной памятью (640 Кбайт) всегда входит расширенная (extended) память, максимальный объем которой определяется шириной адресной шины процессора и для процессоров Intel составляет 4 Гбайт. Реально на персональных компьютерах устанавливается расширенная память объема 32...128 Мбайт; с каждым годом эта величина неуклонно увеличивается. Поскольку функционирование расширенной памяти подчиняется "спецификации расширенной памяти" (Extended Memory Specification, сокращенно – XMS), то и саму память часто называют XMS-памятью. Как уже отмечалось выше, доступ к расширенной памяти осуществляется в защищенном режиме, поэтому для MS-DOS, работающей в реальном режиме, расширенная память недоступна.

Однако в составе DOS имеется драйвер HIMEM.SYS, поддерживающий расширенную память, т. е. позволяющий ее использовать, хотя и ограниченным образом. Первые 64 Кбайт расширенной памяти, точнее, 64 Кбайт–16 байт с адресами от 100000h до 10FFEFh носят специальное название области старшей памяти (High Memory Area, HMA). Эта область замечательна тем, что к ней можно обратиться в реальном режиме

работы процессора, если определить сегмент, начинающийся в самом конце мегабайтового адресного пространства, с адреса FFFF0h, и разрешить использование адресной линии A20. Первые 16 байт этого сегмента заняты ПЗУ, а область со смещениями 0010h...FFFFh можно в принципе использовать под программы и данные. MS-DOS позволяет загружать в HMA (директивой файла CONFIG.SYS DOS=HIGH) значительную часть самой себя, в результате чего занятая системой область стандартной памяти существенно уменьшается. Старшую память обслуживает тот же драйвер HIMEM.SYS, поэтому загрузка DOS в HMA возможна, только если этот драйвер установлен.

Как видно из приведенного выше рисунка, часть адресного пространства верхней памяти, не занятая расширениями BIOS и видеопамятью, оказывается свободной. Эти свободные участки можно использовать для адресации расширенной памяти (не всей, а лишь той ее части, объем которой совпадает с общим объемом свободных адресов верхней памяти).

Отображение расширенной памяти на свободные адреса старшей выполняет драйвер EMM386.EXE, а сами участки верхней памяти, "заполненные" расширенной, называются блоками верхней памяти (Upper Memory Blocks, UMB). MS-SOS позволяет загружать в UMB устанавливаемые драйверы устройств (например, драйвер расширения консоли ANSI.SYS или драйвер компакт-дисков MTMCDAI.SYS), а также любые резидентные программы, как системные, так и прикладные. Загрузка в UMB драйверов осуществляется директивой файла CONFIG.SYS DEVICEHIGH (вместо директивы DEVICE), а загрузка резидентных программ – командой DOS LOADHIGH. Для того чтобы блоки верхней памяти стали доступны, должен быть установлен драйвер EMM386.EXE, а в состав файла CONFIG.SYS включена директива DOS=UMB. При переносе DOS в HMA, а драйверов и резидентных программ в UMB объем свободной памяти для загружаемых программ может быть увеличен приблизительно до 615 Кбайт (см. рис. 22.2, где приведено распределение памяти на конкретной вычислительной системе).

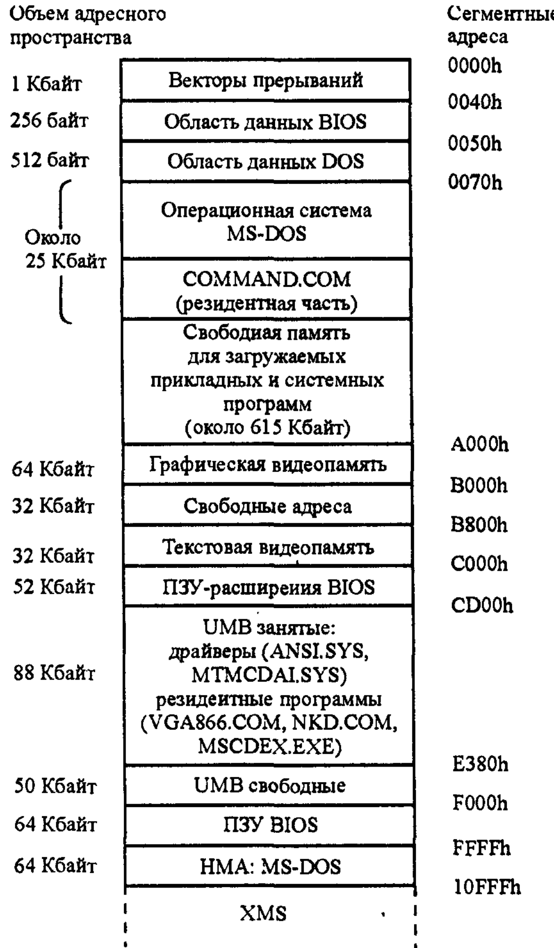


Рис. 22.2. Пример распределения адресного пространства при использовании расширенной памяти

Статья 23. **Система ввода-вывода**

Система ввода-вывода, т. е. комплекс средств обмена информацией с внешними устройствами, является важнейшей частью архитектуры процессора и машины в целом. К системе ввода-вывода можно отнести и способы подключения к системной шине различного оборудования, и процедуры взаимодействия процессора с этим оборудованием, и команды процессора, предназначенные для обмена данными с внешними устройствами.

Непрерывное совершенствование микропроцессоров и стремление максимально повысить производительность всей вычислительной системы привело к существенному усложнению внутренней организации компьютеров: повышению разрядности шин, появлению внутренних быстродействующих магистралей обмена данными, использованию кеш-буферов для ускорения обмена с памятью и дисками и пр. Если, однако, отвлечься от важных с точки зрения производительности, но несущественных для программиста деталей, то логическую схему современного компьютера можно представить традиционным образом, в виде системной шины (магистрали), к которой подключаются сам микропроцессор и все устройства компьютера (рис. 23.1).

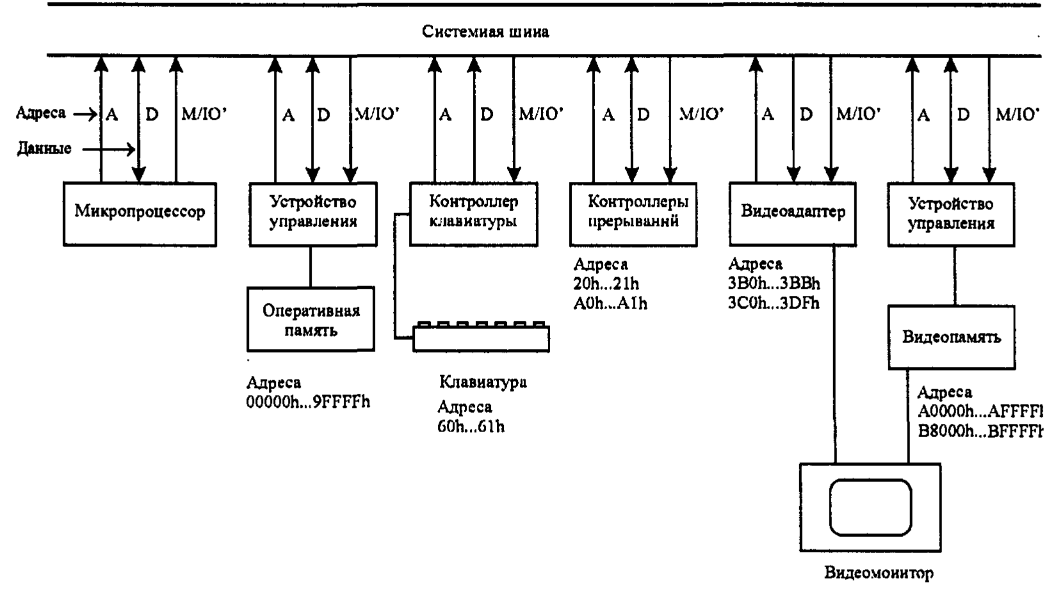


Рис. 23.1. Подключение устройств компьютера к системной шине

Системная шина представляет собой, в сущности, набор линий – проводов, к которым единообразно подключаются все устройства компьютера. В более широком плане в понятие системной шины включают электрические и логические характеристики сигналов, действующих на линиях шины, их назначение, а также правила взаимодействия этих сигналов при выполнении тех или иных операций на шине – то, что обычно называют протоколом обмена информацией. Сигналы, распространяющиеся по шине, доступны всем подключенным к ней устройствам, и в задачу каждого

устройства входит выбор предназначенных ему сигналов и обеспечение реакции на них, соответствующей протоколу обмена.

Процессор связан с системной шиной большим количеством линий (практически всеми своими выводами), из которых нас будут интересовать только линии трех категорий: набор линий адресов, набор линий данных и один из сигналов управления, носящий название M/IO' (M – "IO с отрицанием"). Последний сигнал, строго говоря, имеется только среди выходных сигналов микропроцессора, а на системную шину приходят производные от этого сигнала, образованные как комбинации сигнала M/IO' с управляющими сигналами записи и чтения. Однако суть дела от этого не изменяется, и для простоты мы опустили эти подробности.

Процессор, желая записать данное по некоторому адресу в памяти, выставляет на линии адресов требуемый адрес, а на линии данных – данное. Устройство управления памятью расшифровывает поступивший адрес и, если этот адрес принадлежит памяти, принимает с линий данных поступившее данное и заносит его в соответствующую ячейку памяти. Описанная процедура отражает выполнение процессором команды типа

```
mov    mem,AX
```

где mem – символическое обозначение ячейки памяти, принадлежащей сегменту данных программы.

Если процессор, выполняя "обратную" команду типа

```
mov    AX,mem
```

должен прочитать данное из памяти, он выставляет на линии адресов требуемый адрес и ожидает поступления данных. Устройство управления памятью, расшифровав поступивший адрес и убедившись в наличии такого адреса в памяти, отыскивает в памяти требуемую ячейку, считывает из нее данное и выставляет его на линии данных. Процессор снимает данное с шины и отправляет его в указанный в команде регистр.

Описанные процедуры записи и чтения справедливы не только по отношении к памяти; для всех остальных устройств компьютера они выглядят точно так же. За каждым устройством закреплена определенная группа адресов, на которые оно должно отзываться. Обнаружив свой адрес на магистрали, устройство, в зависимости от заданного процессором направления передачи данных, либо считывает с магистрали поступившие данные, либо, наоборот, устанавливает имеющиеся в нем данные на магистраль.

Из рис. 23.1 видно, что все устройства компьютера можно разбить на две категории. Представителем одной категории является видеопамять, входящая в видеосистему компьютера. Устройство управления видеопамятью настроено на две группы адресов, которые как бы продолжают адреса, относящиеся к оперативной памяти. Действительно, адрес последнего байта оперативной памяти составляет 9FFFFh, а уже следующий адрес, A0000h является адресом первого байта графической видеопамяти. Графическая видеопамять занимает 64 Кбайт адресного пространства до адреса AFFFFh (реально немного меньше, но в плане рассматриваемого вопроса это не имеет значения). Текстовая видеопамять расположена на некотором расстоянии от графической и занимает 32 Кбайт начиная с адреса B8000h. Таким образом, адреса оперативной и видеопамяти разнесены и не перекрываются.

Ко второй категории устройств можно отнести все устройства, адреса которых перекрываются адресами оперативной памяти. Например, за контроллером клавиатуры закреплены два адреса: 60h и 61h. По адресу 60h выполняется чтение кода нажатой

клавиши, а адрес 61h используется для управления работой контроллера. Оба эти адреса имеются в оперативной памяти, и, таким образом, возникает проблема распознавания устройства, к которому происходит обращение. Аналогичная ситуация наблюдается и со многими другими устройствами компьютера. Например, контроллер прерываний, служащий для объединения сигналов прерываний ото всех устройств компьютера и направлсния их на единственный вход прерывания микропроцессора, управляется через два адреса. Поскольку в состав машины всегда включают два контроллера, для них выделяются две пары адресов. Во всех компьютерах типа IBM PC контроллерам прерываний назначаются адреса 20h...21h и A0h...A1h, которые также отвсчают и некоторым байтам опсративной памяти.

Проблема идентификации устройств с перекрывающимися адресами имеет два аспекта: аппаратный и программный. Идентификация устройств на системной шине осуществляется с помощью сигнала M/IO', которой генерируется процессором в любой операции записи или чтения. Однако значение этого сигнала зависит от категории адресуемого устройства. При обращении к оперативной или видсопамяти процессор устанавливает значение сигнала M/IO' = 1 (M обозначает memory, память). При обращении к остальным устройствам этот сигнал устанавливается в 0 (IO обозначает in-out, ввод-вывод, и если IO с отрицанием равно нулю, то IO равно единице, что олицетворяет операцию ввода-вывода). В то же время все устройства, подключенные к шине, анализируют значение сигнала M/IO'. При этом оперативная и видеопамять отзываются на операции чтения-записи на шине, только если они сопровождаются значением M/IO' = 1, а остальные устройства воспринимают сигналы магистрали только при значении M/IO' = 0. Таким образом осуществляется аппаратное разделенис устройств типа памяти и устройств ввода-вывода.

Программное разделение устройств реализуется с помощью двух наборов команд процессора – для памяти и для устройств ввода-вывода. В первую группу команд входят практически все команды процессора, с помощью которых можно обратиться по тому или иному адресу – команды пересылки mov и movs, арифметических действий add, mul и div, сдвигов rol, ror, sal и sar, анализа содержимого test и многие другие. Вторую группу команд образуют специфические команды ввода-вывода. В МП 86 их всего две – команда ввода in и команда вывода out. При выполнении команд первой группы процессор автоматически генсрирует M/IO' = 1; при выполнении команд in и out процессор устанавливаст сигнал M/IO' = 0.

Таким образом, при обращении к оперативной и видеопамяти программист может использовать все подходящие по смыслу команды процессора, при этом, работая, например, с видеопамятью, можно не только засылать в нее (или получать из нее) данные, но и выполнять прямо в видеопамяти любые арифметические, логические и прочие опсрации.

Обращаться же к контроллсрам тех или иных устройств (и между прочим, к видеоадаптеру), допустимо только с помощью двух команд – in и out. Арифметические операции или анализ данных в устройстве невозможен. Необходимо сначала прочитать в процсссор данное из внешнего устройства и лишь затем выполнять над ним требуемую операцию.

Наличие двух категорий адресов устройств дает основание говорить о существовании двух адресных пространств – пространства памяти, куда входит опсративная память, видеопамять и ПЗУ, и пространства ввода-вывода (пространства портов), куда

входят адреса остальной аппаратуры компьютера. При этом если объем адресного пространства памяти составляет 1 Мбайт (а в защищенном режиме 4 Гбайт), то адресное пространство портов гораздо меньше – его размер составляет всего 64 Кбайт. Эта величина определяется форматом команд ввода-вывода. Адрес адресуемого порта должен быть записан в регистр DX (и ни в какой другой), и, таким образом, максимальное значение этого адреса составляет величину FFFFh. Реально из 64 Кбайт адресного пространства портов используется лишь очень малая часть. (Практические вопросы программирования через общее с памятью адресное пространства и через пространство портов будут рассмотрены в последующих статьях этой книги.)

Статья 24. **Видеопамять и ее программирование**

До сих пор для вывода на экран результатов работы программы мы пользовались системными средствами, главным образом функциями DOS с номерами 09h и 40h. Однако в некоторых случаях использование системных средств невозможно. Так, в обработчиках аппаратных прерываний (например, от клавиатуры или таймера) запрещен вызов функций DOS. Причина такого запрета заключается в том, что аппаратное прерывание может прийти в любой момент времени, в частности когда выполняется какая-либо функция DOS. Само по себе прерывание на некоторое время системных программ особой опасности не представляет, однако нельзя, прервав на полдороге выполнение системных программ, начать их выполнять с начала. Промежуточные результаты первого выполнения будут в этом случае затерты. Общие методы (очень своеобразные!) решения этой проблемы будут рассмотрены в разделе 3 этой книги; в настоящей статье мы остановимся лишь на частной проблеме вывода символьной информации на экран без обращения к средствам DOS или BIOS. Речь будет идти о прямом обращении к видеопамяти компьютера, минуя какие-либо системные средства. В дальнейшем, при рассмотрении примеров обработчиков прерываний, этот метод вывода информации на экран нам очень пригодится.

Заметим, что на примере вывода текстовых данных в видеопамять будет рассмотрен чрезвычайно важный вопрос обращения к ячейкам памяти с известными физическими адресами. До сих пор мы работали только с ячейками внутри нашей программы. Эти ячейки могут располагаться как в сегменте данных, так и в сегменте команд; в исходном тексте программы им даются некоторые имена, по которым и происходит обращение к ним в программных строках. Однако видеопамять не входит в нашу программу и у ее ячеек нет никаких имен. В таких случаях обращение к памяти осуществляется по заданным физическим адресам. Для работы с видеопамятью необходимо знать, где в адресном пространстве она находится и какова его организация. Поэтому рассмотрим прежде всего некоторые элементы видеосистемы машин типа IBM PC.

Текстовая память включает 8 видеостраниц и занимает в адресном пространстве компьютера (за пределами обычной памяти) 32 Кбайт от сегментного адреса B800h (см. рис. 22.1). Начинается она с видеостраницы 0, адрес которой совпадает с адресом всей видеопамяти. Каждая страница занимает 4 Кбайт; таким образом, страница 1 начинается с сегментного адреса B900h, страница 2 – с адреса BA00h и т. д. Вся видеопамять простирается до границы (сегментной) C000h.

При включении компьютера активной (видимой) становится видеостраница 0. Смена видеостраниц осуществляется вызовом функции 05h прерывания 10h BIOS.

Любой код, записываемый в видеопамять, сразу же отображается на экране в виде цветного символа на одном из знакомест. Каждый символ занимает в видеопамяти поле из 2 байт (рис. 24.1). Младшие (четные) байты всех полей отводятся под коды ASCII отображаемых символов, старшие (нечетные) байты – под их атрибуты. Соответствие значений атрибутов цветам приведено в табл. 24.1.

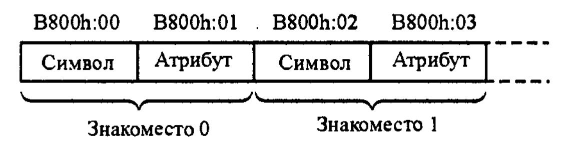


Рис. 24.1 Логическая организация текстовой видеопамяти

Таблица 24.1. Коды цветов стандартной цветовой палитры EGA

Код	*Цвет*	*Код*	*Цвет*
0h	Черный	8h	Серый
1h	Синий	9h	Голубой
2h	Зеленый	0Ah	Салатовый
3h	Бирюзовый	0Bh	Светло-бирюзовый
4h	Красный	0Ch	Розовый
5h	Фиолетовый	0Dh	Светло-фиолетовый
6h	Коричневый	0Eh	Желтый
7h	Белый	0Fh	Ярко-белый

Двухбайтовые коды символов записываются в видеопамять в том порядке, в каком они должны появляться на экране: первые 80 2-байтовых полей соответствуют первой строке экрана, вторые 80 полей – второй строке и т. д. Таким образом, переход на следующую строку экрана определяется не управляющими кодами возврата каретки и перевода строки, а размещением кодов символов в другом месте видеопамяти, в полях, соответствующих следующей строке. Вообще при формировании изображения непосредственно в видеопамяти, в обход программ DOS и BIOS, все управляющие коды ASCII теряют свои управляющие функции и отображаются в виде соответствующих им символов. Трактовка же, например, кода ASCII 9 как символа табуляции или кода ASCII 10 как символа перевода строки выполняется программами DOS, которые в данном случае не активизируются.

Для того чтобы из программы получить доступ к видеопамяти, надо занести в один из сегментных регистров данных ее сегментный адрес. После этого, задавая те или иные смещения, мы сможем выполнять запись в любые места видеопамяти.

Пример 24.1. Прямое программирование видеопамяти

```
;Настроим сегментный регистр ES на страницу 0 видеопамяти
        mov     AX,0B800h     ;(1)Сегментный адрес видеопамяти
        mov     ES,AX         ;(2)Загрузим его в ES
        mov     BX,0          ;(3)Смещение к началу экрана
        mov     SI,80*2*12+40*2;(4)Смещение к центру экрана
        mov     DI,80*2*25-2;(5)Смещение к концу экрана
        mov     word ptr ES:[BX],0F01h;(6)Рожицу на экран
        mov     word ptr ES:[SI],3130h;(7)Нолик на экран
        mov     word ptr ES:[DI],0E40Fh;(8)Большую звездочку на экран
        mov     AH,01h        ;(9)Останов программы
        int     21h           ;(10)для наблюдения результата
```

Программа начинается с занесения в сегментный регистр ES сегментного адреса видеопамяти, т. е. ее физического адреса, деленного на 16. Поскольку непосредственная запись адреса в сегментный регистр запрещена, эта операция осуществляется через промежуточный регистр AX (предложения 1 и 2). Для задания смещения можно воспользоваться базовыми и индексными регистрами. Учитывая, что в каждая строка экрана содержит 80 символов, а символ требуст 2 байт, выражение 80*2*12+40*2 (предложсние 4) описывает смещение от начала видеопамяти точно к центру экрана, а выражение 80*2*25–2 (предложение 5) – к последнему знакоместу экрана. Засылаемая в видеопамять информация в данном примере указывается в виде непосредственных операндов. В предложении 6 знак рожицы (код 01h) с атрибутом ярко-белый на черном фоне (код 0Fh) пересылается командой mov в самое начало видеопамяти; в предложении 7 символ '0' (код 30h) с атрибутом синий на бирюзовом фоне (код 31h) записывается в центр видеостраницы, а в предложении 8 знак большой звездочки (код 0Fh) с атрибутом красный на желтом фонс (код E4h) выводится в самый конец экрана.

В конце программы предусмотрена остановка до нажатия клавиши, чтобы выводимый на экран после завершения текущей программы запрос DOS не разрушил построенного нами изображения.

Как правило, в видеопамять требуется записать не один-два символа, а значительно больший объем данных, вплоть до полной видеостраницы. Обычно выводимый на экран информационный кадр формируется заранее в буфере пользователя, располагающемся в сегменте данных программы. Таким образом, вывод на экран сводится к пересылке содержимого программного буфера в видеопамять.

Как уже отмечалось, команда mov не может осуществлять пересылку из памяти в память. Для этого предусмотрены специальные строковые команды movsb и movsw (см. статью 16). Следует только помнить, что при работе с видеопамятью выводимые символы должны пересылаться не подряд, а через байт, в четные байты видеопамяти. В процессе начальной загрузки компьютера в байты атрибутов всех видеостраниц записывается число 07, что соответствует белым символам на черном фоне. Это дает нам возможность заносить в память только символы текста (через байт), которые будут черно-белыми. Если же требуется вывести цветной текст, в видеопамять следует пересылать 2-байтовые комбинации символ + атрибут. Примеры 24.2 и 24.3 иллюстрируют соответственно то и другое.

Пример 24.2. Запись строки в видеопамять

```
;Настроим сегментный регистр DS на сегмент данных программы,
;а регистр ES на страницу 0 видеопамяти
        ...
;Перешлем в видеобуфер строку символов
;Для этого сначала настроим регистры SI, DI и CX
        mov    SI,offset msg;SI=смещение источника
        mov    DI,80*2*12+37*2;DI=смещение приемника
        mov    CX,msglen   ;CX=число пересылаемых байтов
        cld                ;Сброс DF - вперед
rep     movsb              ;Пересылка в цикле
;Остановим программу
        ...
;Поля данных
msg     db 10h,0Eh,'T',0E0h,'e',0E0h,'c',0E0h,'т',0E0h,11h,0Eh
msglen=$-msg
```

Поскольку в видеопамяти коды ASCII отображаемых символов перемежаются с их атрибутами, пересылаемую в видеопамять текстовую строку приходится формировать так же. При этом для каждого символа можно установить свой атрибут. Однако в целом такой метод описания текстовой строки слишком громоздок. В тех случаях, когда все символы строки имеют один и тот же атрибут, удобнее включение атрибутов в строку выполнять программно. Типичный алгоритм такого рода показан в примере 24.3.

Пример 24.3. Формирование строки для записи в видеопамять цветного текста

```
;Настроим сегментные регистры ES и DS
        ...
        mov    DI,80*2*5   ;Начальное смещение на экране
;Будем переносить байт за байтом на экран, вставляя
;между кодами ASCII байты атрибута
        mov    CX,msglen   ;Длина обрабатываемой строки
        mov    SI,offset msg;Смещение исходной строки
        cld                ;Вперед
        mov    AH,31h      ;Атрибут - синий по бирюзовому
load:   lodsb              ;Загрузили в AL очередной символ
        stosw              ;Символ+атрибут из AX в видеопамять
        loop   load        ;Повторять msglen раз
;Остановим программу
        ...
;Поля данных
msg db 'Обращение к функциям DOS и BIOS осуществляется '
db 'с помощью механизма программных прерываний. Настроив нужным'
db ' образом регистры общего назначения процессора и выполнив к'
db 'оманду программного прерывания INT с соответствующим номером'
db ', пользователь активизирует требуемую функцию DOS или BIOS.'
msglen=$-msg
```

Обратите внимание на то, что команда lodsb будет в каждом шаге цикла смещать адресацию на 1 байт, т. е. к следующему символу выводимой строки, а команда stosw, выводя в видеопамять целое слово (символ + атрибут), будет смещать адресацию на 2 байта, перемещаясь к следующему знакоместу.

Наконец, если раскрашивать текст нет необходимости, можно воспользоваться удобной парой команд lodsb/stosb, поместив их в цикл (пример 24.4). Здесь после каждой команды stosb придется выполнять команду инкремента индексного регистра приемника DI, чтобы обеспечить запись символов в нечетные байты видеопамяти.

Пример 24.4. Вывод в видеопамять текста с атрибутом по умолчанию

```
;Настроим регистры DS и ES
        mov    SI,offset msg
        mov    DI,160*24   ;Нижняя строка экрана
        mov    CX,5        ;Длина текста
oho:    lodsb              ;Заберем символ в AL
        stosb              ;Выведем на экран
        inc    DI          ;К следующему знакоместу
        loop   oho         ;Цикл по всем символам строки msg
;Остановим программу
        ...
;Поля данных
msg     db 'Стоп!'
```

Статья 25. **Система прерываний**

Система прерываний любого компьютера является его важнейшей частью, позволяющей быстро реагировать на события, обработка которых должна выполнятся немедленно: сигналы от машинных таймеров, нажатия клавиш клавиатуры или мыши, сбои памяти и пр. Рассмотрим в общих чертах компоненты этой системы.

Сигналы аппаратных прерываний, возникающие в устройствах, входящих в состав компьютера или подключенных к нему, поступают в процессор не непосредственно, а через два контроллера прерываний, один из которых называется ведущим, а второй – ведомым (рис. 25.1). В прежних моделях машин контроллеры представляли собой отдельные микросхемы; в современных компьютерах они входят в состав многофункциональной микросхемы периферийного контроллера.

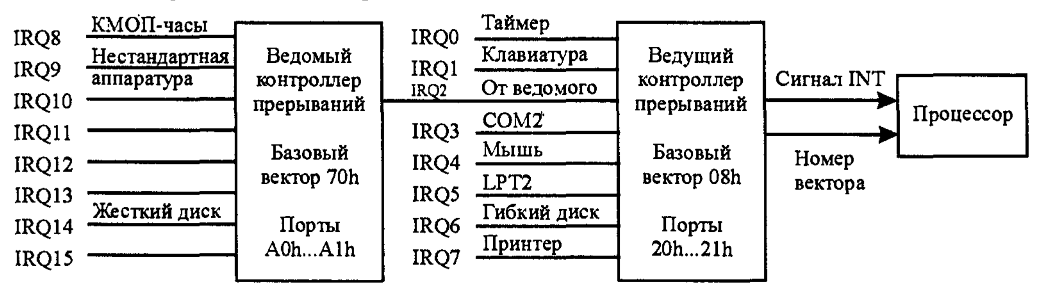


Рис. 25.1. Аппаратная организация прерываний

Два контроллера используются для увеличения допустимого числа внешних устройств. Дело в том, что каждый контроллер прерываний может обслуживать сигналы лишь от восьми устройств. Для обслуживания большего количества устройств контроллеры можно объединять, образуя из них веерообразную структуру. В современных машинах устанавливают два контроллера, увеличивая тем самым возможное число входных устройств до 15 (7 у ведущего и 8 у ведомого контроллера).

К входным выводам IRQ1...IRQ7 и IRQ8...IRQ15 (IRQ – это сокращение от Interrupt Request, запрос прерывания) подключаются выводы устройств, на которых возникают сигналы прерываний. Выход ведущего контроллера подключается ко входу INT микропроцессора, а выход ведомого – ко входу IRQ2 ведущего. Основная функция контроллеров – передача сигналов запросов прерываний от внешних устройств на единственный вход прерываний микропроцессора. При этом, кроме сигнала INT, контроллеры передают в микропроцессор по линиям данных номер вектора, который образуется в контроллере путем сложения базового номера, записанного в одном из его регистров, с номером входной линии, по которой поступил запрос прерывания. Номера базовых векторов заносятся в контроллеры автоматически в процессе начальной загрузки компьютера. Для ведущего контроллера базовый вектор всегда равен восьми, для ведомого – 70h. Таким образом, номера векторов, закрепленных за аппаратными прерываниями, лежат в диапазонах 8h...Fh и 70h...77h. Очевидно, что номера векторов аппаратных прерываний однозначно связаны с номерами линий, или уровнями IRQ, а через них – с конкретными устройствами компьютера. На рис. 24.1 обозначены некоторые из стандартных устройств компьютера, работающих в режиме прерываний.

Процессор, получив по линии INT сигнал прерывания, выполняет последовательность стандартных действий, которую можно назвать процедурой прерывания. Подчеркнем, что здесь идет речь лишь о реакции самого процессора на сигналы прерываний, а не об алгоритмах обработки прерываний, предусматриваемых пользователем в обработчиках прерываний.

Объекты вычислительной системы, принимающие участие в процедуре прерывания, и их взаимодействие показаны на рис. 25.2.

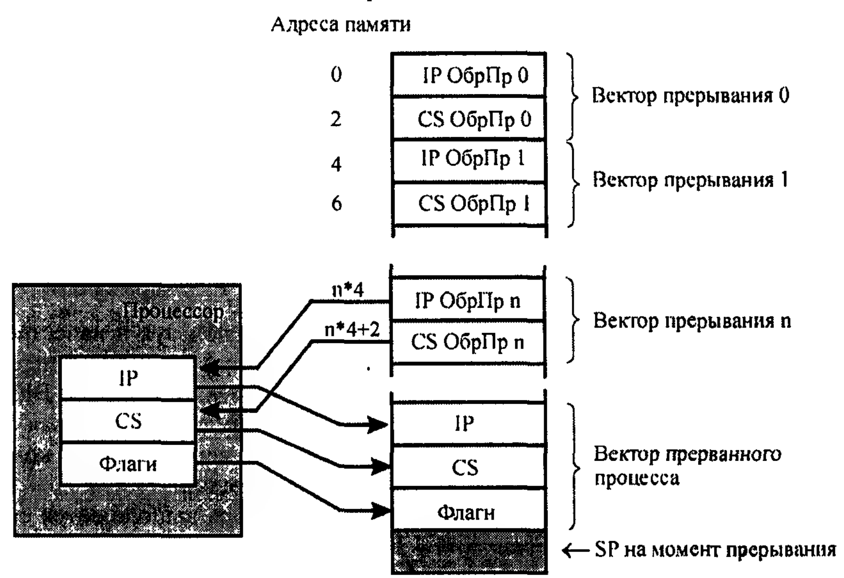


Рис. 25.2. Процедура обслуживания прерывания

Как уже отмечалось ранее (см. статью 22), самое начало оперативной памяти от адреса 00000h до 003FFh отводится под векторы прерываний – 4-байтовые области, в которых хранятся адреса обработчиков прерываний (ОбрПр на рис. 25.2). В два старших байта каждого вектора записывается сегментный адрес обработчика, в два младших – смещение точки входа в обработчик. Векторы, как и соответствующие им прерывания, имеют номера, причем вектор с номером 0 располагается начиная с адреса 0, вектор 1 – с адреса 4, вектор 2 – с адреса 8 и т. д. Вектор с номером *n* занимает, таким образом, байты памяти от *n**4 до *n**4+3. Всего в выделенной под векторы области памяти помещается 256 векторов.

Получив сигнал на выполнение процедуры прерывания с определенным номером, процессор сохраняет в стеке выполняемой программы текущее содержимое трех регистров процессора: регистра флагов, CS и IP. Два последних числа образуют полный адрес возврата в прерванную программу. Далее процессор загружает CS и IP из соответствующего вектора прерываний, осуществляя тем самым переход на обработчик прерываний, связанный с этим вектором.

Обработчик прерываний всегда заканчивается командой iret (interrupt return, возврат из прерывания), выполняющей обратные действия – извлечение из стека сохраненных там слов и помещение их назад в регистры IP, CS и FLAGS. Это приводит к возврату в основную программу в ту самую точку, где она была прервана.

В действительности запросы на обработку прерываний могут иметь различную природу. Помимо описанных выше аппаратных прерывания от периферийных устройств, называемых часто внешними, имеются еще два типа прерываний: внутренние и программные.

Внутренние прерывания возбуждаются цепями самого процессора при возникновении одной из специально оговоренных ситуаций, например при выполнении операции деления на нуль или при попытке выполнить несуществующую команду. За каждым из таких прерываний закреплен определенный вектор, номер которого известен процессору. Например, за делением на 0 закреплен вектор 0, а за неправильной командой – вектор 6. Если процессор сталкивается с одной из таких ситуаций, он выполняет описанную выше процедуру прерывания, используя закрепленный за этой ситуацией вектор прерывания.

Чрезвычайно важным типом прерываний являются программные прерывания. Они вызываются командой int с числовым аргументом, который рассматривается процессором как номер вектора прерывания. Если в программе встречается, например, команда

```
int 13h
```

то процессор выполняет ту же процедуру прерывания, используя в качестве номера вектора операнд команды int. Программные прерывания применяются в первую очередь для вызова системных обслуживающих программ – функций DOS и BIOS. С командой int 21h вызова DOS мы уже сталкивались ранее и будем встречаться еще многократно. В дальнейшем будут также приведены примеры использования команды int для вызова прикладных обработчиков программных прерываний.

Важно подчеркнуть, что описанные действия процессора выполняются совершенно одинаково для всех видов прерываний – внутренних, аппаратных и программных, хотя причины, возбуждающие процедуру прерывания, имеют принципиально разную природу.

Всего, как уже было сказано, в памяти отводится место под 256 векторов прерываний. Некоторые из них практически не используются; к другим, наоборот, приходится обращаться едва ли не в каждой программе. Для того чтобы дать представление о составе таблицы векторов, приведем краткую выборку из нее, перечислив векторы, представляющие наибольший интерес для пользователя (табл. 25.1).

Таблица 25.1. Назначение некоторых векторов прерываний

Номера векторов	*Адреса векторов*	*Назначение векторов*
00h	00000h	Внутреннее прерывание деления на нуль
01h	00004h	Внутреннее прерывание пошагового режима (используется отладчиками)
02h	00008h	Немаскируемое прерывание (отказ питания, ошибка памяти)
03h	0000Ch	Программное прерывание от команды int без числового параметра
04h	00010h	Внутреннее прерывание от команды into при фиксации переполнения
05h	00014h	Аппаратное прерывание от нажатия клавиши PrintScreen (печать экрана)

06h	00018h	Внутреннее прерывание недопустимого кода операции
08h	00020h	Аппаратное прерывание от системного таймера
09h	00024h	Аппаратное прерывание от клавиатуры
0Bh	0002Ch	Аппаратное прерывание от последовательного порта COM2
0Ch	00030h	Аппаратное прерывание от последовательного порта COM1 (обычно от мыши)
0Dh	00034h	Аппаратное прерывание от параллельного порта LPT2 (обычно свободно)
0Eh	00038h	Аппаратное прерывание от гибкого диска
0Fh	0003Ch	Аппаратное прерывание от параллельного порта (принтера; обычно свободно)
10h	00040h	Программы BIOS обслуживания видеосистемы
13h	0004Ch	Программы BIOS обслуживания дисков
14h	00050h	Программы BIOS обслуживания последовательного порта
16h	00058h	Программы BIOS обслуживания клавиатуры
17h	0005Ch	Программы BIOS обслуживания принтера
1Ah	00068h	Программы BIOS обслуживания часов реального времени
1Ch	00070h	Вектор для прикладной обработки прерываний от системного таймера
1Dh	00074h	Адрес таблицы видеопараметров, используемой программами BIOS
1Eh	00078h	Адрес таблицы параметров дискеты, используемой программами BIOS
21h	00084h	Диспетчер функций DOS
22h	00088h	Адрес возврата в родительский процесс после завершения текущего
23h	0008Ch	Программа DOS завершения процесса по команде Ctrl+C
24h	00090h	Адрес обработчика DOS критической ошибки
25h	00094h	Программа DOS абсолютного чтения диска
26h	00098h	Программа DOS абсолютной записи на диск
2Fh	000BCh	Программное прерывание, используемое для связи с резидентными программами
33h	000CCh	Программы обслуживания мыши, реализуемые в драйвере мыши
4Ah	00128h	Вектор для прикладной обработки прерываний от будильника (часов реального времени)
60h...66h	00180h... ...00198h	Свободные векторы для использования прикладными программами
70h	001C0h	Аппаратное прерывание от будильника (часов реального времени)
71h	001C4h	Аппаратное прерывание от подключенной к компьютеру нестандартной аппаратуры
76h	001D8h	Аппаратное прерывание от жесткого диска

Как видно из приведенной таблицы, векторы прерываний можно разбить на следующие группы:

- векторы внутренних прерываний процессора (01h, 02h и др.);
- векторы аппаратных прерываний (08h...0Fh и 70h...77h);
- программы BIOS обслуживания аппаратуры компьютера (10h, 13h, 16h и др.);
- программы DOS (21h, 22h, 23h и др.);
- адреса системных таблиц BIOS (1Dh, 1Eh и др.).

Системные программы, адреса которых хранятся в векторах прерываний, в большинстве своем являются всего лишь диспетчерами, открывающими доступ к большим группам программ, реализующих системные функции. Так, программа BIOS обслуживания видеосистемы (вектор 10h) включает функции смены видеорежима, управления курсором, задания цветовой палитры, загрузки шрифтов и многие другие. Особенно характерен в этом отношении вектор 21h, через который, как мы уже знаем, осуществляется вызов практически всех функций DOS, используемых в прикладных программах: ввода с клавиатуры и вывода на экран, обслуживания файлов, каталогов и дисков, управления памятью и процессами, службы времени и т. д. Однако ряд функций DOS реализуются через другие векторы. Так, для прямого доступа к гибким и жестким дискам (не по имени файла, а по номеру сектора на диске) предусмотрены специальные программные векторы 25h и 26h, а обработка введенной с клавиатуры команды Ctrl+C выполняется с помощью вектора 23h.

Статья 26. **Контроллер прерываний и его программирование**

Как было показано в предыдущей статье, сигналы прерываний от периферийного оборудования компьютера поступают в процессор не непосредственно, а через систему из двух контроллеров прерываний. Для того чтобы написать программу обработки прерываний от какого-либо устройства, необходимо быть знакомым с особенностями функционирования и программирования контроллера прерываний.

Рассмотрим внутреннюю структуру контроллера (для определенности возьмем ведущий контроллер). Логически в ней можно выделить 4 основных узла: регистр входных запросов, регистр маски, схему приоритетов и регистр обслуживаемых запросов (рис. 26.1). Все эти узлы 8-битовые, по одному биту на каждый входной сигнал.

Сигнал запроса прерывания IRQ от устройства (на рис. 26.1 это IRQ1, т. е. сигнал от клавиатуры), поступает на вход регистра запросов и устанавливает в 1 соответствующий бит этого регистра. Далее на пути сигнала стоит регистр маски, программируемый через порт 21h. Значение 0 в бите маски разрешает прохождение сигнала, значение 1 – запрещает. Пройдя через маску, сигнал поступает на схему анализа приоритетов.

При стандартной настройке схемы анализа приоритетов (она программируется посылкой определенных команд через порт 20h) приоритеты сигналов IRQ снижаются по мере роста номера сигнала, т. е. максимальным приоритетом обладает сигнал IRQ0, минимальным – IRQ7.

Ведомый контроллер всегда подключается ко входу IRQ2 ведущего, поэтому все приоритеты ведомого контроллера располагаются между приоритетами уровней IRQ1 и IRQ3 ведущего и образуется такая цепочка приоритетов:

IRQ0...IRQ1 --- IRQ8...IRQ15 --- IRQ3...IRQ7
Высший приоритет — Приоритеты ведомого — Низший приоритет

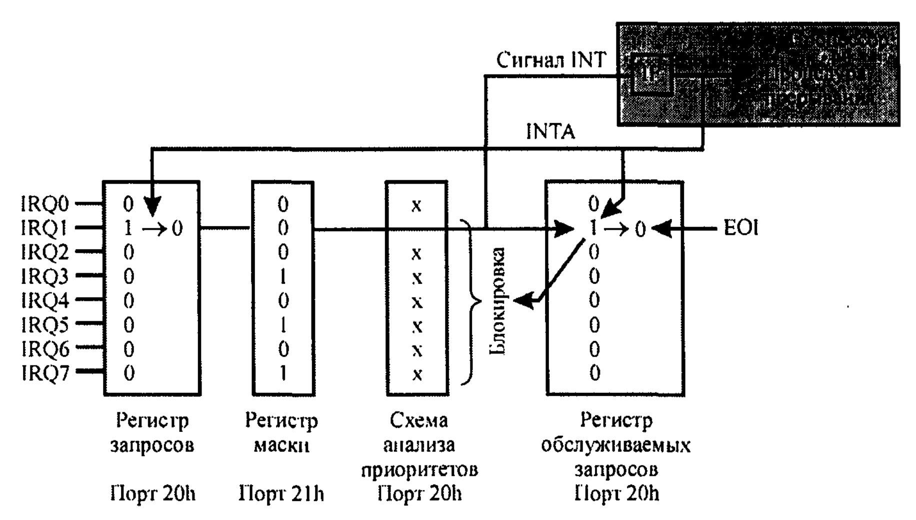


Рис. 26.1. Логическая структура контроллера прерываний

Приоритеты уровней прерываний не имеют никакого значения, пока прерывания поступают редко и не накладываются друг на друга. Вопрос о приоритетах становится важным только в том случае, если очередной сигнал прерывания приходит в тот момент, когда еще не закончено выполнение программы обработки предыдущего прерывания. Алгоритм обработки таких наложенных прерываний определяет программист путем соответствующего построения обработчика прерывания.

Пройдя через схему анализа приоритетов, сигнал запроса прерывания поступает на вход регистра обслуживаемых запросов и дает разрешение на установку в 1 его бита (однако не устанавливает его). Одновременно сигнал поступает на вход INT микропроцессора. Микропроцессор регистрирует поступление сигнала INT лишь в том случае, если установлен флаг разрешения прерываний IF в регистре флагов. Таким образом, сброс флага IF командой cli запрещает все аппаратные прерывания (не затрагивая программные).

Микропроцессор, получив сигнал INT, отвечает на него выходным сигналом INTA (INTerrupt Acknowledge, подтверждение прерывания), который поступает в контроллер прерываний и выполняет там два действия: устанавливает бит регистра обслуживаемых запросов, разрешенный сигналом запроса прерывания, и сбрасывает аналогичный бит регистра запросов. Таким образом, запрос, для которого началась процедура обслуживания его микропроцессором, переводится в разряд обслуживаемых. Начиная с этого момента на тот же вход контроллера прерываний может прийти следующий сигнал прерывания от устройства. Он, правда, какое-то время не будет обслуживаться, но, по крайней мере, не пропадет, а запомнится в регистре запросов и будет ждать своей очереди на обслуживание контроллером и процессором.

Микропроцессор одновременно с посылкой в контроллер прерываний сигнала INTA сбрасывает флаг IF в регистре флагов, запрещая все аппаратные прерывания.

Прерывания останутся запрещенными до выполнения пользователем команды sti или до установки флага IF каким-либо другим способом.

Установка 1 в бите регистра обслуживаемых запросов воздействует на схему анализа приоритетов. Установленный бит блокирует в схеме анализа приоритетов все уровни прерываний начиная с текущего и ниже. Таким образом, если не принять специальных мер, даже после завершения программы обработчика прерывания все прерывания данного и более низких приоритетов останутся заблокированными. Сброс бита регистра обслуживаемых запросов осуществляется засылкой кода 20h в порт 20h для ведущего контроллера и в порт A0h для ведомого. Этот код получил название команды или приказа EOI (End Of Interrupt, конец прерывания). Приказ конца прерывания должен возбуждаться в любом обработчике прерываний.

Исходя из изложенного, в программе обработки прерывания можно выделить три участка, которые показаны на рис. 26.2 для конкретного случая прихода запроса прерывания уровня 1.

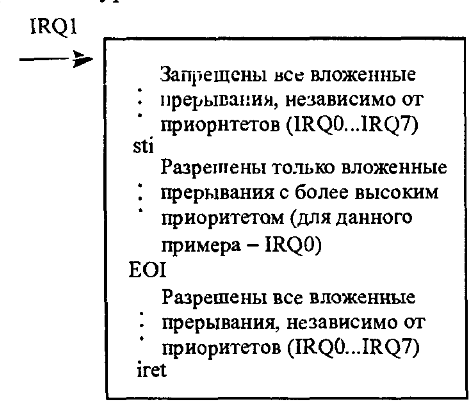


Рис. 26.2. Обобщенная структура программы обработчика прерываний

Поскольку процессор, получив сигнал INT, сбрасывает флаг IF, при входе в обработчик все прерывания оказываются запрещенными и программа не может быть прервана внешними сигналами. Команда sti, если она будет выполнена в обработчике прерываний, установит флаг IF и разрешит прохождение запросов прерываний в процессор. Однако все уровни прерываний, начиная с текущего, остаются заблокированными в контроллере. В результате работа обработчика может быть прервана только при поступлении запроса прерывания более высокого приоритета (в рассматриваемом примере – IRQ0). Такая ситуация называется вложенным прерыванием.

Выполнение в обработчике команд, реализующих приказ конца прерывания EOI, снимает блокировку в контроллере, и начиная с этого момента запрос прерывания любого уровня прервет выполнение обработчика. Особенно неприятной может оказаться ситуация, когда обработчик прерывается сигналом запроса прерывания того же уровня. Это приводит к тому, что программа обработчика, не дойдя до конца, опять начинает выполняться с самого начала, т. е. произойдет повторный вход в программу. Для того чтобы такое явление не нарушило работоспособность системы, обработчик прерываний должен быть написан по определенным правилам, обеспечивающим его реентерабельность (повторную входимость). Лучше, однако, избегать возникновения такой ситуации.

Практически структуру программы обработки прерывания выбирают исходя из конкретных условий (рис. 26.3).

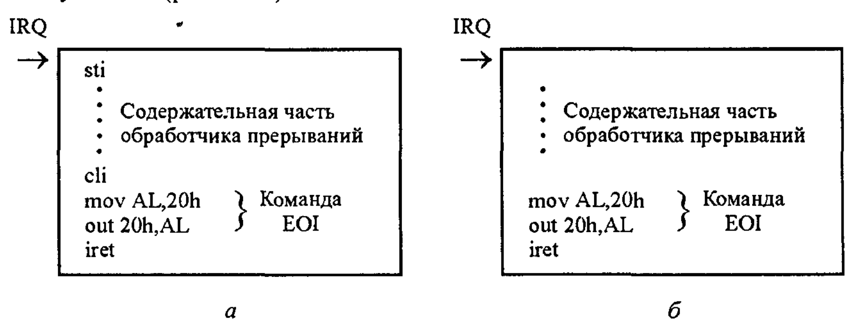


Рис. 26.3. Типичные структуры обработчиков прерываний:
а – при разрешенных вложенных прерываниях; б – при запрещенных вложенных прерываниях

Часто в самом начале обработчика прерываний выполняют команду sti, чтобы не задерживать обработку прерываний от более приоритетных устройств (в частности, таймера). Приказ конца прерывания EOI посылается в контроллер в самом конце программы, перед завершающей командой iret, с тем чтобы полностью исключить вложенные прерывания от запросов того же уровня (рис. 26.3, *а*). Однако сигнал прерывания того же (или более низкого) уровня может прийти между командой out 20h,AL и командой iret. Поскольку блокировка нижележащих уровней в контроллере уже снята, возникнет вложенное прерывание и повторный вход в ту же программу. Чтобы избежать этого, перед командой EOI выполняют команду запрета всех прерываний cli. В результате вложенные прерывания запрещаются до выхода из обработчика. Можно подумать, что прерывания останутся запрещенными навсегда, однако это не так. Команда iret восстанавливает не только адрес возврата в регистрах CS и IP, но и содержимое регистра флагов на момент прерывания. Если при этом содержимом регистра флагов возникло прерывание, значит, флаг IF был установлен. Таким образом, команда iret в обработчике аппаратного прерывания всегда возвращает установленное состояние флага IF, т. е. разрешает прерывания.

Между прочим, отсюда следует, что в обработчике прерываний вообще может отсутствовать команда sti. В этом случае программа обработчика будет выполняться при запрещенных прерываниях (всех уровней), а разрешены прерывания (опять же все) будут после выполнения завершающей команды iret (рис. 26.3, *б*).

Запросы на прерывания, поступающие в ведущий контроллер (уровни IRQ0... ...IRQ7), блокируют только ведущий контроллер. Однако запросы на прерывания, поступающие в ведомый контроллер (уровни IRQ8...IRQ15), блокируют не только уровни низших приоритетов в ведомом контроллере, но и уровни IRQ2...IRQ7 в ведущем контроллере. Поэтому в обработчиках аппаратных прерываний уровней 8...15 следует предусматривать посылку команд EOI в оба контроллера:

```
mov   AL,20h
out   20h,AL
out   A0h,AL
```

Рассмотрим пару простых примеров программирования контроллера прерываний.

Пример 26.1. Запрещение прерываний от таймера

```
        in      AL,21h          ;Прочитаем текущую маску
        or      AL,1            ;Установим добавочно бит 0
        out     21h,AL          ;Пошлем в регистр маски
```

Выполнение примера 26.1 приведет к остановке системного таймера. Это легко проконтролировать с помощью часов программы Norton Commander или с помощью команды DOS TIME.

Чтобы не нарушать работу машины, после программы 26.1 следует выполнить программу 26.2.

Пример 26.2. Разрешение прерываний от таймера

```
        in      AL,21h          ;Прочитаем текущую маску
        and     AL,0FEh         ;Сбросим выборочно бит 0
        out     21h,AL          ;Пошлем в регистр маски
```

Таким же образом можно, например, запретить прерывания от контроллера гибких дисков и для контроля попробовать обратиться к дискете, хотя бы с помощью программы Norton Commander.

Обычно программирование контроллера прерываний в прикладных программах сводится к посылке сигнала EOI и маскированию (или размаскированию) в случае необходимости отдельных уровней прерываний. Однако в некоторых случаях приходится выполнять более серьезное вмешательство в настройку контроллера. Например, перед переходом в защищенный режим следует изменить базовые векторы обоих контроллеров.

Для смены базового вектора требуется выполнить полностью процедуру инициализации контроллера, которая состоит из ряда так называемых слов команд инициализации (СКИ), посылаемых в строгой последовательности друг за другом. В зависимости от конфигурации микропроцессорного устройства, в котором используется контроллер прерываний, значения команд инициализации могут различаться, однако в компьютерах типа IBM PC контроллеры программируются всегда единообразно.

Первое слово инициализации, СКИ1, посылаемое для ведущего контроллера в порт 20h, а для ведомого – в порт A0h, всегда имеет значение 11h.

Остальные три слова инициализации посылаются во вторые управляющие порты контроллеров: 21h – для ведущего контроллера и A1h – для ведомого.

Второе слово инициализации, СКИ2 задает базовый вектор. Как уже отмечалось, в системе MS-DOS базовый вектор ведущего контроллера равен восьми, ведомого – 70h. При использовании аппаратных прерываний в защищенном режиме контроллеры приходится перепрограммировать; так, в системах Windows ведущему контроллеру назначается базовый вектор 50h, ведомому – 57h.

Третье слово инициализации, СКИ3 выглядит по-разному для ведущего и ведомого контроллеров. Для ведущего в слове СКИ3 устанавливаются в 1 те биты, которые соответствуют входам IRQ, подключенным к ведомым контроллерам. Биты, соответствующие входам IRQ, подключенным к периферийным устройствам, сбрасываются. Для всех компьютеров типа IBM PC ведомый контроллер подсоединяется ко входу 2 ведущего, поэтому в СКИ3 должен быть установлен бит 2 и сброшены остальные биты, что соответствует числу 4.

Для ведомого контроллера в слове инициализации СКИ3 указывается номер входа IRQ ведущего контроллера, к которому подключен данный ведомый (которых, как уже

отмечалось, в принципе может быть несколько). В компьютерах ведомый контроллер подсоединяется ко входу 2 ведущего, поэтому слово СКИ3 равно двум.

Выше уже отмечалось, что в процессе обслуживания поступившего сигнала прерывания контроллер блокирует все нижележащие уровни. В слове СКИ4 указывается, выполняется ли снятие этой блокировки автоматически, или для этого требуется команда EOI. В компьютерах используется режим с командой EOI; в этом случае СКИ4 равно единице.

Таким образом, для инициализации обоих контроллеров прерываний (для стандартной конфигурации компьютера) следует выполнить такие последовательности команд:

```
;Инициализация ведущего контроллера прерываний
        mov     DX,20h          ;Первый порт контроллера
        mov     AL,11h          ;СКИ1
        out     DX,AL
        jmp     $+2             ;Задержка
        inc     DX              ;Второй порт контроллера
        mov     AL,8            ;СКИ2: базовый вектор
        out     DX,AL
        jmp     $+2             ;Задержка
        mov     AL,4            ;СКИ3: ведомый подключен
                                ;к уровню 2
        out     DX,AL
        jmp     $+2             ;Задержка
        mov     AL,1            ;СКИ4: требуется EOI
        out     DX,AL
;Инициализация ведомого контроллера прерываний
        mov     DX,A0h          ;Первый порт контроллера
        mov     AL,11h          ;СКИ1
        out     DX,AL
        jmp     $+2             ;Задержка
        inc     DX              ;Второй порт контроллера
        mov     AL,70h          ;СКИ2: базовый вектор
        out     DX,AL
        jmp     $+2             ;Задержка
        mov     AL,2            ;СКИ3: ведомый подключен
                                ;к уровню 2
        out     DX,AL
        jmp     $+2             ;Задержка
        mov     AL,1            ;СКИ4: требуется EOI
        out     DX,AL
```

После каждой команды посылки кода в порт контроллера предусмотрена небольшая задержка (команда jmp на следующий байт программы), чтобы аппаратура контроллера успела воспринять посылаемую команду. Конкретная величина задержки зависит от соотношения скоростей работы процессора и контроллера; часто оказывается, что в задержке нет необходимости.

Инициализировав оба контроллера, следует установить в них маски прерываний. Конкретное значение масок зависит от аппаратной конфигурации компьютера. Типичными значениями являются B8h для ведущего контроллера и 9Dh для ведомого.

При отладке взаимодействия с компьютером аппаратуры, работающей в режиме прерываний, приходится анализировать содержимое внутренних регистров контроллера прерываний. Для этого существуют специальные команды контроллера.

Если в порт 20h контроллера прерываний послать код 0Ah, то разрешается чтение входного регистра контроллера – регистра запросов. Чтение (в том числе неоднократное) осуществляется через порт 20h. Читая содержимое IRR, можно определить, на ка-

кие входы контроллера поступают сигналы аппаратуры. При этом надо иметь в виду, что регистрация процессором (процессором, не контроллером!) сигнала прерывания приводит к сбросу запроса соответствующего уровня в регистре запросов. Поэтому наблюдение запросов во входном регистре следует выполнять либо при замаскированных, либо при запрещенных прерываниях.

Код 0Bh, посланный в тот же порт 20h, разрешает чтение регистра обслуживаемых запросов. Чтение (в том числе неоднократное) осуществляется через порт 20h. Таким образом можно установить, проходит ли сигнал прерывания в процессор (поскольку установка битов регистра обслуживаемых запросов выполняется сигналом INTA, поступающим из процессора в контроллер только после регистрации им сигнала прерывания).

Статья 27. **Системные таймеры**

Современные компьютеры оснащаются двумя подсистемами таймеров, параллельно отсчитывающими текущее время. Один таймер расположен в микросхеме с низким потреблением энергии (КМОП-микросхеме), которая при выключении питания компьютера продолжает работать, получая энергию от встроенного в компьютер аккумулятора. Этот таймер обычно называют часами реального времени; помимо кварцевого генератора и систем управления он содержит внутреннюю память, в которой хранятся и наращиваются значения текущей даты и времени. Обновление времени осуществляется каждую секунду, причем эта операция выполняется на аппаратном уровне, не затрагивая ни процессор, ни основную оперативную память.

Другой таймер (его часто называют системным) работает, как и все остальные узлы компьютера, только когда компьютер включен. Он вырабатывает сигналы с частотой приблизительно 18,206 Гц, вызывающие аппаратные прерывания уровня 0 (вектор 8). Обработчик этих прерываний, входящий в систему BIOS, с каждым прерыванием увеличивает на единицу содержимое двухсловной ячейки с адресом 40h:6Ch, расположенной в области данных BIOS. В процессе начальной загрузки компьютера программа BIOS считывает показания часов реального времени (часы, минуты и секунды) и, преобразовав их в число секунд, истекшее от начала текущих суток, умножает эту величину на 18,206, чтобы получить текущее время, выраженное в числе тактов системного таймера. Эта величина заносится в ячейку с адресом 40h:6Ch. Выполняемый в дальнейшем инкремент этой ячейки поддерживает в ней, пока компьютер включен, правильное текущее время. Именно из этой ячейки черпается информация о текущем времени программами DOS: функцией 2Ch или командой TIME.

Рассмотрим более подробно структуру и возможности программного управления обоих таймеров.

Подсистема часов реального времени включает контроллер и небольшой блок памяти объемом 64 байт. Первые 14 байт этой памяти используются для отсчета времени; остальные 50 байт хранят информацию о конфигурации системы. Подсистема обеспечивает следующие функции:

- отсчет текущего времени с точностью до 1 с. Текущее время (отдельно год, месяц, день, минута и секунда) хранится в памяти микросхемы в двоично-десятичном коде (BCD) и может быть прочитано оттуда в любой момент времени;

- работу программируемого будильника, который в установленное время дает сигнал прерывания (называемого сигнальным) по линии IRQ 8, закрепленной за вектором 70h. Будильник можно запрограммировать на выдачу сигнального прерывания в заданное время суток (каждые сутки до сброса будильника); в заданное время каждого часа (раз в час); в заданную секунду каждой минуты (раз в минуту);
- режим периодических прерываний по той же линии IRQ8, частоту которых можно программно настраивать в пределах 2 Гц...8 кГц;
- хранение данных о конфигурации системы (объем базовой и расширенной памяти, типы магнитных дисков и пр.);
- хранение пароля в зашифрованном виде.

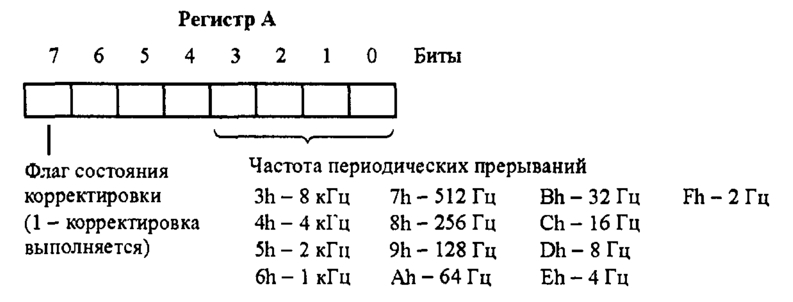
Назначение отдельных байтов КМОП-памяти (нумеруемых от 0 до 3Fh) приведено в табл. 27.1.

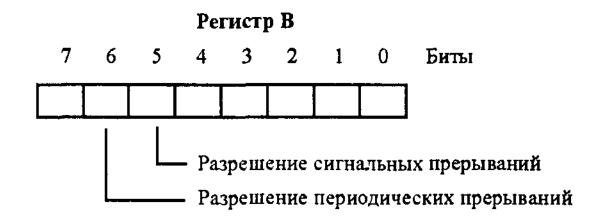
Таблица 27.1. Адресное пространство памяти КМОП-микросхемы

Адрес поля	*Число байтов*	*Назначение*
00h	1	Секунды в BCD
01h	1	Секунды будильника в BCD
02h	1	Минуты в BCD
03h	1	Минуты будильника в BCD
04h	1	Часы в BCD
05h	1	Часы будильника в BCD
06h	1	День недели (может отсутствовать)
07h	1	Число месяца в BCD
08h	1	Месяц (январь – 1 и т. д.) в BCD
09h	1	Год, две младшие цифры в BCD
0Ah	1	**Регистр А**
0Bh	1	**Регистр В**
0Ch	1	**Регистр С**
0Dh	1	**Регистр D**
0Eh	1	Байт диагностирования
0Fh	1	Байт кода сброса процессора
10h	1	Типы жестких дисков (если < 15)
11h	1	Зарезервировано
12h	1	Типы дискет
13h	1	Зарезервировано
14h	1	Состав установленного оборудования
15h	2	Объем базовой памяти в килобайтах
17h	2	Объем расширенной памяти в килобайтах
19h	1	Тип первого жесткого диска (если > 15)
1Ah	1	Тип второго жесткого диска (если > 15)
1Bh	14	Зарезервировано
2Eh	2	Контрольная сумма байтов 10h...2Dh
30h	2	Объем расширенной памяти в килобайтах

32h	1	Год, две старшие цифры в BCD
33h	1	Системная информация
34h	12	Зарезервировано

Байты с номерами Ah, Bh, Ch и Dh выполняют функции управляющих регистров контроллера. Назначение их битов можно найти в технической документации; на рис. 27.1 приведены лишь сведения, полезные для прикладного программиста.





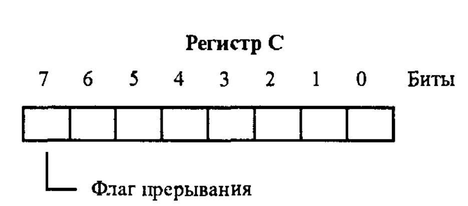


Рис. 27.1. Управляющие регистры часов реального времени

Байты КМОП-памяти читаются и записываются через порты 70h...71h. Обращение к памяти осуществляется в два этапа: сначала в порт 70h посылается номер требуемого байта памяти, затем через порт 71h осуществляются чтение (командой in) или запись (командой out) байта памяти.

Пример 27.1. Чтение байта памяти из КМОП-микросхемы

```
mov   AL,17h    ;Будем читать байт 17h
out   70h,AL    ;
in    AL,71h    ;Младший байт объема XMS
mov   BL,AL     ;Сохраним в BL
mov   AL,18h    ;Будем читать байт 18h
out   70h,AL    ;
in    AL,71h    ;Старший байт объема XMS
mov   BH,AL     ;BX=объем XMS
```

После выполнения приведенного фрагмента в регистре BX окажется число, на 1024 меньшее, чем полный объем памяти (в килобайтах), установленной на компьютере. Таким же образом можно получить, например, объем базовой памяти (байты 15h и 16h) или содержимое регистров A, B, C и D.

Описанным выше способом можно обращаться ко всем байтам КМОП-памяти, за исключением первых десяти. Дело в том, что КМОП-микросхема один раз в секунду выполняет корректировку текущего времени, а также проверку состояния будильника. На время корректировки участок КМОП-памяти, относящийся к часам, календарю и будильнику, отключается от системной магистрали и становится недоступным для программного обращения. На это же время устанавливается бит 7 регистра A. Поэтому при чтении или записи байтов 00h...09h необходимо сначала дождаться сброса бита 7 этого регистра, что знаменует окончание цикла корректировки, и лишь затем выполнять обращение к требуемым байтам КМОП-памяти. В примере 27.2 показана процедура чтения текущей секунды (байт 0 КМОП-памяти).

Пример 27.2. Чтение байта КМОП-памяти с ожиданием окончания цикла корректировки

```
        mov     AL,0Ah          ;Будем читать регистр A
        out     70h,AL
wait1:  in      AL,71h          ;Регистр A в AL
        test    AL,80h          ;Бит 7 установлен?
        jnz     wait1           ;Да, повторим проверку
        mov     AL,0h           ;Корректировки нет, будем
        out     70h,AL          ;читать байт 0 (секунды)
        in      AL,71h          ;Текущая секунда в AL
```

Часы реального времени работают от внутреннего кварцевого генератора, частота которого подобрана так, что сигналы на его выходе (после пересчета) следуют с интервалом точно 1 с. Эти сигналы используются для отсчета текущего времени в байтах часов и календаря КМОП-памяти. Помимо постоянного пересчета, обеспечивающего частоту 1 Гц, в КМОП-схему включен еще узел настраиваемого пересчета, выходные сигналы которого поступают на линию IRQ8, инициируя периодические прерывания через вектор 70h. Коэффициент пересчета и, соответственно, частоту периодических прерываний можно программно настраивать, изменяя содержимое битов 0...3 регистра A (см. рис. 27.1). Для разрешения и запрещения периодических прерываний используется бит 6 регистра B.

Установка будильника осуществляется занесением требуемого времени "побудки" в байты секунд (1), минут (3) и часов (5) КМОП-памяти. При записи в эти байты необходимо предусматривать ожидание окончания циклов корректировки, как это было показано в примере 27.1. Схемы таймера периодически сравнивают текущее время с временем, записанным в байтах будильника, и при достижении равенства вырабатывают сигнал сигнального прерывания на той же линии IRQ8. Для разрешения и запрещения сигнальных прерываний используется бит 5 регистра B.

Если "законные" числа установлены во всех трех байтах будильника, сигнал прерывания будет формироваться в указанное время каждые сутки. Однако программа может установить в одном или нескольких байтах будильника "безразличный" код – любое число от C0h до FFh. Если безразличный код установлен в байте часов будильника, то сигнальные прерывания вырабатываются каждый час. При установке безразличного кода в байтах часов и минут прерывания вырабатываются каждую минуту. Наконец, при наличии безразличного кода во всех трех байтах будильника прерывание вырабатывается каждую секунду. Особенность такого режима заключается в том, что

прерывания формируются не просто с заданной частотой, а в заданный момент каждой минуты или каждого часа. Например, если в байте секунд будильника записано число 30h, а в байтах минут и часов – C0h, то прерывания будут возникать точно на 30-й секунде каждой минуты.

При обработке прерываний от КМОП-микросхемы необходимо иметь в виду, что сигналы, предназначенные для возбуждения прерываний (как сигнальных, так и периодических), поступают на вход контроллера прерываний не непосредственно, а через разряд 7 регистра C, выполняющий функции флага прерываний. Сигнал прерывания устанавливает этот флаг, что, в свою очередь, приводит к установке на входе IRQ8 контроллера прерываний активного уровня. Программа обработки прерываний обязана сбрасывать флаг прерываний, иначе на входе контроллера прерываний будет поддерживаться активный уровень и дальнейшее поступление сигналов прерываний окажется заблокированным. Сброс флага прерываний осуществляется чтением регистра C. Заметим, что периодические прерывания от КМОП-микросхемы не используются в стандартной конфигурации компьютера и не поддерживаются ни DOS, ни BIOS. При разработке обработчика периодических прерываний (например, для компьютерной системы автоматизации) необходимо программировать КМОП-микросхему на физическом уровне, через порты 70h и 71h.

Для чтения и изменения показаний часов реального времени предусмотрено прерывание BIOS 1Ah, функции которого обращаются непосредственно к КМОП-памяти и позволяют не только получить или установить дату и время, но и управлять будильником. Пример использования этого прерывания (конкретно – для чтения из КМОП-микросхемы текущего времени) был рассмотрен в программе 19.1.

Перейдем теперь к рассмотрению системного таймера.

Системный трехканальный таймер, обычно входящий в состав многофункциональной микросхемы программируемого периферийного контроллера, выполняет три функции: отсчет системного времени, управление регенерацией динамической памяти и возбуждение звука в динамике компьютера. В последней процедуре участвует один из управляющих портов периферийного контроллера (именно порт 61h). На рис. 27.2 приведена упрощенная схема взаимодействия этих узлов.

Подсистема таймера работает независимо от процессора (и параллельно с ним) от собственного генератора, вырабатывающего сигналы с частотой 1,19318 МГц. Таймер имеет три независимых канала, каждый из которых можно перепрограммировать отдельно от других. Для управления режимами программирования в подсистеме таймера имеется регистр команд, обращение к которому осуществляется через порт 43h. Каждый канал таймера включает счетчик, пересчитывающий сигналы от генератора, и регистр-фиксатор, в который программно заносится число, определяющее коэффициент пересчета счетчика. Фиксаторы каналов таймера адресуются через порты 40h, 41h и 42h соответственно.

При включении компьютера в фиксатор канала 0 заносится максимально возможное число 65535 (FFFFh), в результате чего сигналы на выходе канала 0 имеют частоту 18,2 1/с. Эти сигналы, как уже отмечалось выше, возбуждают прерывания с вектором 08h, которые обрабатываются программой BIOS, осуществляющей отсчет текущего времени.

Прерывания от канала 0 можно использовать для временной синхронизации программы, например для периодического вывода на экран некоторой информации. Эти вопросы будут рассмотрены в разделе 3 настоящей книги.

Канал 1 таймера в программах не используется.

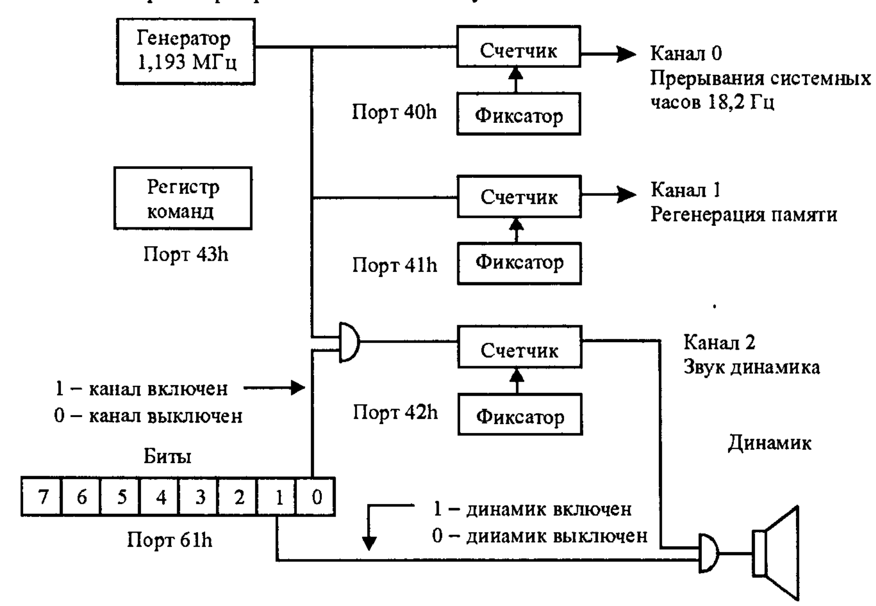


Рис. 27.2. Элементы компьютера, принимающие участие в генерации звука

Выход канала 2 связан с динамиком и используется для генерации звука. Изменяя содержимое фиксатора этого канала, можно изменять частоту сигналов, поступающих на динамик, от 18,2 Гц до 1,19 МГц. Реально для возбуждения звука можно использовать частоты не выше приблизительно 10 кГц.

Программирование всех каналов таймера осуществляется одинаково. В регистр команд (порт 43h) засылается управляющее слово, характеризующее режим работы программируемого канала, после чего в фиксатор канала загружается константа пересчета. Формат управляющего слова приведен на рис. 27.3.

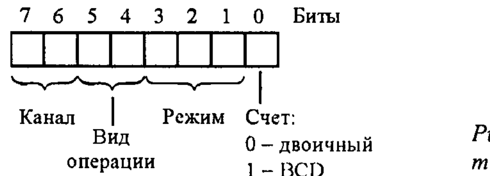


Рис. 27.3. Формат управляющего слова таймера (порт 43h)

Бит 0 управляющего слова определяет способ задания константы пересчета. При нулевом значении бита константа задается в двоичной форме, при единичном – в двоично-десятичной (BCD). Чаще используется двоичный способ задания константы. В биты 1...3 засылается режим работы таймера (всего имеется 6 режимов с кодами 0...5), а в биты 4...5 – один из четырех возможных видов операции. В биты 6...7 заносится номер программируемого канала (0...2). В табл. 27.2 приведен перечень режимов, а в табл. 27.3 – возможные виды операций.

Таблица 27.2. Режимы работы каналов таймера

Код режима	Описание
0	Формирование одиночного прямоугольного сигнала заданной длительности
1	Формирование одиночного сигнала заданной длительности с аппаратным запуском
2	Генерация кратковременных периодических сигналов
3	Генерация периодических прямоугольных сигналов со скважностью 2 ("меандр")
4	Формирование кратковременного сигнала с заданной задержкой
5	Формирование кратковременного сигнала с заданной задержкой с аппаратным запуском

Таблица 27.3. Виды операций при чтении-записи фиксаторов каналов таймера

Код вида операции	Описание
0	Чтение "на лету"
1	Чтение-запись младшего байта фиксатора
2	Чтение-запись старшего байта фиксатора
3	Чтение-запись сначала младшего, затем старшего байта фиксатора

Следует отметить, что таймер, используемый в персональных компьютерах, разрабатывался как многофункциональное устройство для получения в микропроцессорных устройствах программно-управляемых задержек и генерации времязадающих сигналов. В нем предусмотрено большое количество режимов работы, конкретное функционирование которых к тому же зависит от того, какие сигналы подаются на различные входы схемы таймера. В компьютерах таймер подключается вполне определенным образом и большая часть перечисленных выше режимов не используется.

Прикладной программист может столкнуться с программированием канала 0 для организации временной синхронизации программы или внешних процессов, а также канала 1 для генерации звуковых сигналов. Оба канала работают в режиме 3 генерации периодических сигналов со скважностью 2, частота которых определяется содержимым фиксатора. Для занесения в фиксатор коэффициента пересчета входной частоты используется вид операции 3, когда 16-разрядная константа пересчета загружается в фиксатор двумя последовательными командами out. Первое обращение к порту фиксатора заполняет младший байт фиксатора, а следующее обращение – старший.

Приведем несколько примеров программного управления каналами таймера.

Пример 27.3. Повышение частоты системного таймера

```
main    proc
        mov   AL,36h      ;(1)Канал 0, режим 3, вид операции 3
        out   43h,AL      ;(2)Управляющее слово в порт
        mov   AX,6535     ;(3)Константа для частоты 182 Гц
        out   40h,AL      ;(4)Ее младший байт в фиксатор
        mov   AL,AH       ;(5)AL=старший байт константы
        out   40h,AL      ;(6)Отправим его в фиксатор
        mov   AX,4C00h    ;(7)Завершим программу
        int   21h
main    endp
```

В предложениях 1 и 2 управляющее слово, которое для данного случая имеет код 36h, засылается в порт 43h регистра команд. В регистр AX загружается константа 6535, обеспечивающая увеличение частоты системного таймера в 10 раз. В соответствии с выбранным кодом вида операции загрузка константы в регистр-фиксатор требует двух команд out. Первая команда out (предложение 4) загружает младший байт константы, следующая за ней команда out с обращением к тому же порту – старший (предложение 6). Частота таймера увеличивается и остается такой после завершения программы. Для восстановления правильного хода системных часов придется либо выполнить еще раз программу 27.1, изменив в ней константу пересчета на 65535, либо перезагрузить компьютер. Результат работы программы нагляднее всего наблюдать с помощью программы Norton Commander, в кадре которого имеется поле для показаний системных часов. После завершения программы двоеточие в этом поле, олицетворяющее ход секунд, начнет мигать в 10 раз быстрее.

Пример 27.3 в предложенной редакции может иметь лишь познавательное значение. Реально, однако, увеличение частоты системных часов может быть весьма полезным, если компьютер входит в автоматизированную измерительную систему, в которой требуется задавать точные временные интервалы. В стандартном режиме таймер отсчитывает время с погрешностью приблизительно 1/18 с, что для многих применений слишком много. Если увеличить частоту работы таймера до, например, 1000 Гц (для чего потребуется загрузить в фиксатор константу 1193180 Гц/1000 Гц = 1193), то отсчитывать с его помощью интервалы времени можно будет с погрешностью 10^{-3} с, если фиксировать прерывания от таймера и пересчитывать их в заданной пропорции. Например, отсчет 10000 прерываний позволит задать интервал времени, равный 10 с, с погрешностью всего 0,01 %. Методика составления обработчиков прерываний, в том числе и от системного таймера, будет рассмотрена в 3-м разделе этой книги.

Следует заметить, что приведенная выше программа будет работать неправильно в сеансе DOS системы Windows 95, так как в этой системе состояние таймера сеанса DOS принудительно восстанавливается сразу же после завершения программы. В NT-подобных системах (как, разумеется, и в чистой DOS) этого явления не наблюдается: системы Windows NT/2000 оставляют состояние перепрограммированного таймера неизменным до конца данного сеанса DOS. С другой стороны, целесообразность запуска такого рода программ в системах Windows вообще сомнительна. Программы управления аппаратурой в реальном времени следует либо выполнять в чистой DOS, либо разрабатывать их специальным образом как приложения Windows.

Рассмотрим теперь простой пример генерации звука с помощью таймера.

Управление каналом 2 таймера и его подключение к динамику осуществляется через порт 61h периферийного контроллера (см. рис. 27.2). Бит 0 порта 61h управляет включением и выключением канала 2 таймера. Пока бит 0 установлен, на выходе канала действуют периодические сигналы заданной фиксатором частоты; при сбросе бита 0 колебания прекращаются. Бит 1 того же порта управляет посылкой в динамик тока. Таким образом, имеются две возможности прекратить звучание тона: запретить работу канала 2 путем сброса бита 0 порта 61h или выключить прохождение через динамик тока путем сброса бита 1 того же порта.

Детали программирования таймера будут ясны из примера 27.4.

Пример 27.4. Управление звуком с помощью таймера

```
.386
text    segment use16
...
;Установим режим таймера
        mov   AL,0B6h     ;(1)Канал 2, режим 3, вид операции 3
        out   43h,AL      ;(2)В регистр команд
;Установим частоту канала 2 таймера
        mov   AX,11930    ;(3)1193000 Гц/11930=100 Гц
        out   42h,AL      ;(4)Младший байт константы в порт
        mov   AL,AH       ;(5)AL=старший байт константы
        out   42h,AL      ;(6)Старший байт константы в порт
;Включим динамик и разрешим таймер
        in    AL,61h      ;(7)Введем содержимое порта 61h
        or    AL,3        ;(8)Установим биты 0 и 1
        out   61h,AL      ;(9)Выведем в порт
;После задержки выключим динамик и запретим таймер
        mov   ECX,50000000;(10)Задержка ~ 1 с
delay:  db    67h         ;(11)Префикс изменения размера адреса
        loop  delay       ;(12)Цикл
        and   AL,11111100b;(13)Сбросим в AL биты 0 и 1
        out   61h,AL      ;(14)Выведем в порт
```

Программа 27.4 возбуждает тон заданной частоты (100 Гц) и продолжительности. После засылки в регистр команд управляющего слова (предложения 1 и 2), а в регистр-фиксатор – константы пересчета (предложения 3...6) выполняется операция разрешения канала 2 таймера и включения динамика. Поскольку порт 61h является многофункциональным (в нем имеются биты управления клавиатурой, памятью и переключателями системной платы), непосредственно заслать в него код, устанавливающий или сбрасывающий отдельные разряды, нельзя – мы тем самым изменим состояние других разрядов. В таких случаях организуется трехступенчатая процедура изменения содержимого порта: чтение из порта, изменение в прочитанном байте требуемых разрядов и запись в порт модифицированного содержимого. В предложениях 7...9 к содержимому порта 61h (неизвестному нам) добавляются 2 младших бита (число 3). Тем самым включается ток динамика и его периодическое переключение от таймера.

Таймер относится к числу аппаратных программируемых элементов компьютера. Все такие элементы программируются только один раз – на этапе начальной загрузки компьютера. Система (MS-DOS) никогда не восстанавливает исходные режимы аппаратуры, да и просто ничего не знает об изменении режима, если его выполнила прикладная программа. Если программно включить звук динамика и завершить программу, динамик будет продолжать гудеть. Поэтому в примере 27.4 после небольшой задержки (предложения 10...12) в сохраненном содержимом регистра AL снова сбрасываются 2 младших бита и эта величина пересылается в порт, выключая звук. Еще раз напоминаем, что величина программной задержки зависит от скорости работы процессора и ее следует подбирать экспериментально. Организация программной задержки с помощью 32-разрядного регистра ECX была описана в статье 8.

Статья 28. **Клавиатура**

Работая на компьютере, пользователю постоянно приходится вводить с клавиатуры команды и данные. Для правильного составления соответствующих фрагментов программы необходимо хорошо представлять, каким образом вводятся, куда попадают

и как обрабатываются символы, вводимые с клавиатуры. Процесс взаимодействия системы с клавиатурой показан на рис. 28.1.

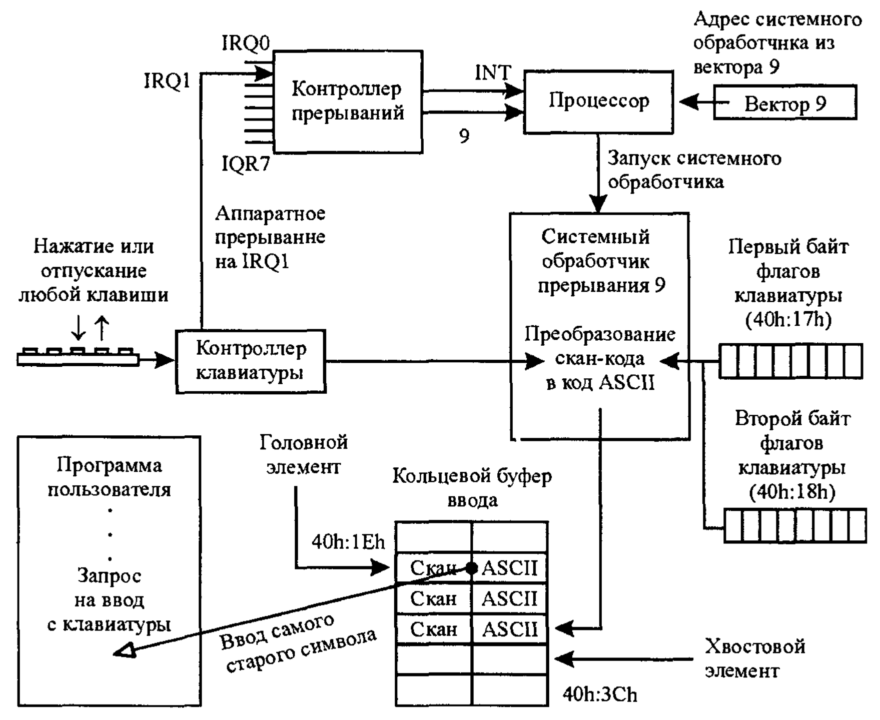
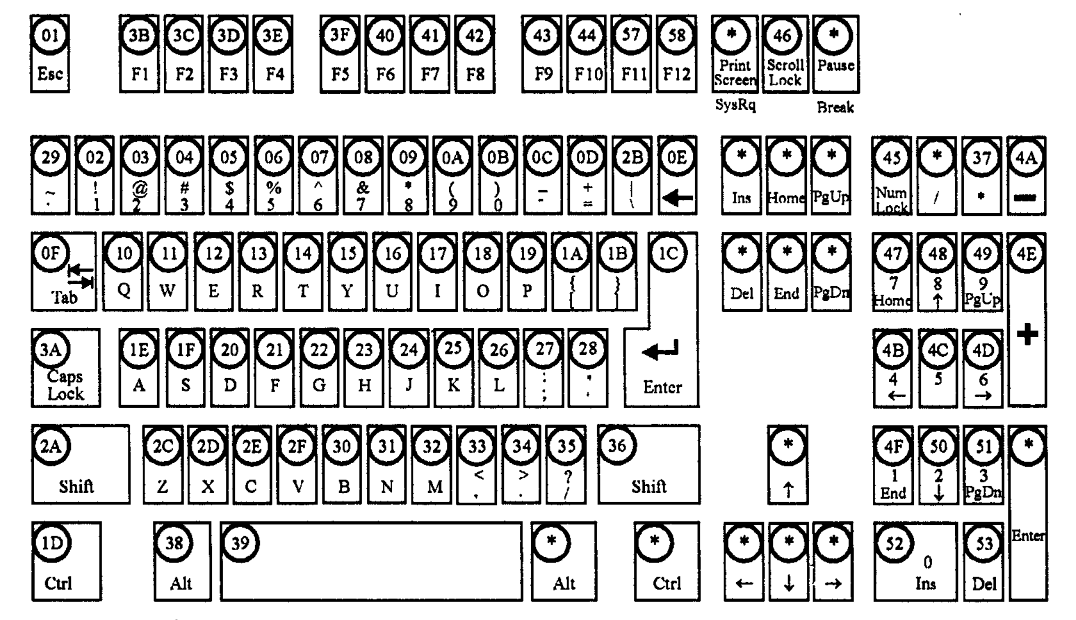


Рис. 28.1. Процесс взаимодействия системы с клавиатурой

Работой клавиатуры управляет специальная электронная схема – контроллер клавиатуры. В его функции входит распознавание нажатой клавиши и помещение закрепленного за ней кода в свой выходной регистр (порт) с номером 60h. Код клавиши, поступающий в порт, называется скан-кодом и является, по существу, порядковым номером клавиши. При этом каждой клавише присвоены два скан-кода, отличающиеся друг от друга на 80h. Один скан-код (меньший, код нажатия) засылается контроллером в порт 60h при нажатии клавиши, другой (больший, код отпускания) – при ее отпускании. Скан-коды клавиш (числа 16-ричные) приведены на рис. 28.2.

Некоторые клавиши и их сочетания генерируют при нажатии не один, а два или даже несколько сигналов прерываний, каждый из которых сопровождается посылкой в порт 60h своего кода. Так, клавиша Ins при нажатии создаст пару скан-кодов: E0h, 52h; клавиша Home – коды E0h, 47h; сочетание клавиш Shift+↑ дает 4 скан-кода: E0h, AAh, E0h, 48h; клавиша Pause (Break) генерирует на каждое нажатие целых 6 скан-кодов: E1h, 1Dh, 45h, E1h, 9Dh, C5h – и т. д. Все такие клавиши помечены на рис. 28.2 звездочками.



* Клавиши с последовательностями скан-кодов

Рис. 28.2. Скан-коды клавиш

Скан-код однозначно указывает на нажатую клавишу, однако по нему нельзя определить, работает ли пользователь на нижнем или верхнем регистре. С другой стороны, скан-коды присвоены всем клавишам клавиатуры, в том числе клавишам Shift, Ctrl, Alt, Caps Lock и др. Таким образом, очевидно, что определение введенного символа должно включать в себя не только считывание скан-кода нажатой клавиши, но и выяснение того, не были ли перед этим нажаты, например, клавиши Shift (верхний регистр) или Caps Lock (фиксация верхнего регистра). Всем этим анализом занимается обработчик прерываний от клавиатуры.

Как нажатие, так и отпускание любой клавиши вызывает сигнал аппаратного прерывания, заставляющий процессор прервать выполняемую программу и перейти на программу системного обработчика прерываний от клавиатуры, входящего в систему BIOS. Поскольку обработчик вызывается через вектор 09, его иногда называют программой int09.

Программа int09, помимо порта 60h, работает еще с двумя областями оперативной памяти: кольцевым буфером ввода, располагаемым по адресам от 40h:1Eh до 40h:3Dh, куда в конце концов помещаются коды ASCII нажатых клавиш, и 2 байтами флагов клавиатуры, находящимися по адресам 40h:17h и 40h:18h. В этих байтах фиксируется состояние управляющих клавиш (Shift, Caps Lock, Num Lock и др.).

Программа int09, получив управление в результате прерывания от клавиатуры, считывает из порта 60h скан-код и анализирует его значение. Если скан-код принадлежит одной из управляющих клавиш и к тому же представляет собой код нажатия, в первом байте флагов клавиатуры устанавливается бит (флаг), соответствующий нажатой клавише. Например, при нажатии правой клавиши Shift в байте флагов устанавливается бит 0, при нажатии левой клавиши Shift – бит 1, при нажатии любой клавиши

Ctrl – бит 2 и т. д. Биты флагов сохраняют свое состояние, пока клавиши (по одиночке или в любых комбинациях) остаются нажатыми. Если управляющая клавиша отпускается, программа int09 получает скан-код отпускания и сбрасывает соответствующий бит в байте флагов. Сказанное вполне справедливо для клавиш Shift, Ctrl или Alt, у которых может быть только два состояния – клавиша нажата и клавиша отпущена. Однако другие управляющие клавиши действуют сложнее. Например, при однократном нажатии клавиши Caps Lock включается режим верхнего регистра клавиатуры (режим прописных букв). При повторном нажатии той же клавиши этот режим выключается. Для программы обычно важно не состояние клавиши Caps Lock (нажата или отпущена), а то, какой режим ею установлен. В первом слове флагов клавиатуры фиксируются режимы всех управляющих клавиш (табл. 28.1), а из второго слова флагов можно получить более детальную информацию о состоянии управляющих клавиш (табл. 28.2).

Таблица 28.1. Назначение битов первого слова флагов клавиатуры (40h:17h)

Номер бита	*Назначение бита*
0	Нажата правая клавиша Shift
1	Нажата левая клавиша Shift
2	Нажата любая клавиша Ctrl
3	Нажата любая клавиша Alt
4	Режим Scroll Lock активен
5	Режим Num Lock активен
6	Режим Caps Lock активен
7	Режим Ins активен

Таблица 28.2. Назначение битов второго слова флагов клавиатуры (40h:18h)

Номер бита	*Назначение бита*
0	Нажата левая клавиша Ctrl
1	Нажата левая клавиша Alt
2	Нажата правая клавиша Ctrl
3	Нажата правая клавиша Alt
4	Нажата клавиша Scroll Lock
5	Нажата клавиша Num Lock
6	Нажата клавиша Caps Lock
7	Нажата клавиша SysRq

При нажатии обычной, не управляющей клавиши программа int09 считывает из порта 60h ее скан-код нажатия и по таблице трансляции скан-кодов в коды ASCII формирует 2-байтовый код, старший байт которого содержит скан-код, а младший – код ASCII. При этом если скан-код характеризует клавишу, то код ASCII определяет закрепленный за ней символ. Поскольку за каждой клавишей закреплено, как правило, не менее двух символов (a и A, 1 и !, 2 и @ и т. д.), то каждому скан-коду соответствуют, как минимум, два кода ASCII. В процессе трансляции программа int09 анализирует состояние флагов, так что если нажата, например, клавиша Q (скан-код 10h, код ASCII буквы Q – 51h, а буквы q – 71h), то формируется 2-байтовый код 1071h, но если клавиша Q нажата при нажатой клавише Shift (смена регистра), то результат трансляции составит 1051h. Тот же код 1051h получится, если при нажатии клавиши Q был включен режим Caps Lock (прописные буквы), однако при

включенном режиме Caps Lock и нажатой клавише Shift образуется код 1071h, поскольку в такой ситуации клавиша Shift на время нажатия переводит клавиатуру в режим нижнего регистра (строчные буквы).

Полученный в результате трансляции 2-байтовый код засылается программой int09 в кольцевой буфер ввода, который служит для синхронизации процессов ввода данных с клавиатуры и приема их выполняемой компьютером программой. Объем буфера составляет 16 слов, при этом коды символов извлекаются из него в том же порядке, в каком они в него поступали. За состоянием буфера следят два указателя. В хвостовом указателе (слово по адресу 40:1Ch) хранится адрес первой свободной ячейки, в головном указателе (40:1Ah) – адрес самого старого кода, принятого с клавиатуры и еще не востребованного программой. Оба адреса представляют собой смещения относительно начала области данных BIOS, т. е. числа от 1Eh до 3Ch. В начале работы, когда буфер пуст, оба указателя – и хвостовой и головной – указывают на первую ячейку буфера.

Программа int09, сформировав 2-байтовый код, помещает его в буфер по адресу, находящемуся в хвостовом указателе, после чего этот адрес увеличивается на 2, указывая опять на первую свободную ячейку. Каждое последующее нажатие на какую-либо клавишу добавляет в буфер очередной 2-байтовый код и смещает хвостовой указатель.

Выполняемая программа, желая получить код нажатой клавиши, должна обратиться для этого к каким-либо системным средствам – функциям ввода с клавиатуры DOS (прерывание 21h) или BIOS (прерывание 16h). Системные программы с помощью драйвера клавиатуры (точнее говоря, объединенного драйвера клавиатуры и экрана, так называемого драйвера консоли с именем CON) считывают из кольцевого буфера содержимое ячейки, адрес которой находится в головном указателе, и увеличивают этот адрес на 2. Таким образом, программный запрос на ввод с клавиатуры фактически выполняет прием кода не с клавиатуры, а из кольцевого буфера.

Хвостовой указатель, перемещаясь по буферу в процессе занесения в него кодов, доходит, наконец, до конца буфера (адрес 40h:3Ch). В этом случае при поступлении очередного кода адрес в указателе не увеличивается, а, наоборот, уменьшается на длину буфера. Тем самым указатель возвращается в начало буфера, затем продолжает перемещаться по буферу до его конца, опять возвращается в начало и так далее по кольцу. Аналогичные манипуляции выполняются и с головным указателем.

Равенство адресов в обоих указателях свидетельствует о том, что буфер пуст. Если при этом программа поставила запрос на ввод символа с клавиатуры, то драйвер консоли будет ждать поступления кода в буфер, после чего этот код будет передан в программу. Если же хвостовой указатель, перемещаясь по буферу в процессе его заполнения, подошел к головному указателю "с обратной стороны" (это произойдет, если оператор нажимает на клавиши, а выполняемая в настоящий момент программа не обращается к клавиатуре), прием новых кодов блокируется, а нажатие на клавиши возбуждает предупреждающие звуковые сигналы.

Если компьютер не выполняет никакой программы, то активной является программа командного процессора COMMAND.COM. Активность COMMAND.COM заключается в том, что он, поставив запрос к DOS на ввод с клавиатуры (с помощью

функции 0Ah прерывания 21h), ожидает ввода с клавиатуры очередной команды пользователя. Как только в кольцевом буфере ввода появляется код символа, функция 0Ah переносит его во внутренний буфер DOS, очищая при этом кольцевой буфер ввода, а также выводит символ на экран. При получении кода клавиши Enter (0Dh) функция 0Ah завершает свою работу, а командный процессор предполагает, что ввод команды закончен, анализирует содержимое буфера DOS и приступает к выполнению введенной команды. При этом командный процессор работает практически лишь с младшими половинами 2-байтовых кодов символов, а именно с кодами ASCII.

Если компьютер выполняет какую-либо программу, ведущую диалог с оператором, то, как уже отмечалось, ввод данных с клавиатуры (а точнее, из кольцевого буфера ввода) и отображение их на экране организует эта программа, обращаясь непосредственно к драйверу BIOS (int 16h) или к соответствующей функции DOS (int 21h). Может случиться, однако, что выполняемой программе не требуется ввода с клавиатуры, а оператор нажал какие-то клавиши. В этом случае вводимые символы накапливаются (с помощью программы int09) в кольцевом буфере ввода и, естественно, не отображаются на экране. Так можно ввести до 15 символов. Когда программа завершится, управление будет передано COMMAND.COM, который сразу же обнаружит наличие символов в кольцевом буфере, извлечет их оттуда и отобразит на экране. Если при этом последовательность введенных символов завершается кодом клавиши Enter, командный процессор воспримет введенную строку как команду и попытается ее выполнить. Такой ввод с клавиатуры называют вводом с упреждением.

До сих пор речь шла о символах и кодах ASCII, которым соответствуют определенные клавиши терминала и которые можно отобразить на экране. Это буквы (прописные и строчные), цифры, знаки препинания и специальные знаки, используемые в программах и командных строках, например [, $, # и др. Однако имеется ряд клавиш, которым не назначены отображаемые на экране символы. Это, например, функциональные клавиши F1...F12; клавиши управления курсором Home, End, PgUp, ←, → и др. При нажатии этих клавиш в кольцевой буфер ввода засылается расширенный код ASCII, в котором младший байт равен нулю, а старший является скан-кодом нажатой клавиши. Расширенные коды ASCII поступают в буфер ввода и в случае нажатия комбинаций управляющих и функциональных клавиш, например Shift+F1, Ctrl+Home (на дополнительной цифровой клавиатуре), Alt+Insert и др. В этом случае, однако, в старший байт расширенного кода ASCII помещается уже не скан-код клавиши, а некоторый код, специально назначенный этой комбинации клавиш. Естественно, этого кода нет среди "обычных" скан-кодов. Например, клавиша F1, скан-код которой равен 3Bh, может генерировать следующие расширенные коды ASCII:

```
F1       3B00h     Ctrl/F1  5E00h
Alt/F1   6800h     Shift/F1 5400h
```

Расширенные коды ASCII широко используются в прикладных программах для управления ходом программы и переключения ее режимов. В табл. 28.3 и 28.4 приведены значения информационных байтов расширенных кодов ASCII клавиш и их комбинаций.

Таблица 28.3. Информационные байты расширенных кодов ASCII функциональных клавиш и их комбинаций с управляющими клавишами

Клавиша	*Информационные байты кодов ASCII клавиш и их сочетаний*			
	Клавиша	*Shift+клавиша*	*Ctrl+клавиша*	*Alt+клавиша*
F1	3Bh	54h	5Eh	68h
F2	3Ch	55h	5Fh	69h
F3	3Dh	56h	60h	6Ah
F4	3Eh	57h	61h	6Bh
F5	3Fh	58h	62h	6Ch
F6	40h	59h	63h	6Dh
F7	41h	5Ah	64h	6Eh
F8	42h	5Bh	65h	6Fh
F9	43h	5Ch	66h	70h
F10	44h	5Dh	67h	71h
F11	85h	87h	89h	8Bh
F12	86h	88h	8Ah	8Ch

Таблица 28.4. Информационные байты расширенных кодов ASCII алфавитно-цифровых и управляющих клавиш в сочетании с клавишей Alt

Клавиша	*Код ASCII*	*Клавиша*	*Код ASCII*	*Клавиша*	*Код ASCII*
~ `	29h	U	16h	B	30h
! 1	78h	I	17h	N	31h
@ 2	79h	O	18h	M	32h
# 3	7Ah	P	19h	< ,	33h
$ 4	7Bh	{ [	1Ah	> .	34h
% 5	7Ch	}]	1Bh	? /	35h
^ 6	7Dh	Enter	1Ch	Ins	A2h
& 7	7Eh	A	1Eh	Home	97h
* 8	7Fh	S	1Fh	Page Up	99h
(9	80h	D	20h	Del	A3h
) 0	81h	F	21h	End	9Fh
-	82h	G	22h	Page Down	A1h
+ =	83h	H	23h	↑	98h
| \	2Bh	J	24h	←	9Bh
Backsp ←	0Eh	K	25h	↓	A0h
Tab	A5h	L	26h	→	9Dh
Q	10h	: ;	27h	Малая цифровая клавиатура	
W	11h	" '	28h	/	A4h
E	12h	Z	2Ch	*	37h
R	13h	X	2Dh	Серый -	4Ah
T	14h	C	2Eh	Серый +	4Eh
Y	15h	V	2Fh	Enter	A6h

Примеры программного обращения к клавиатуре как с помощью различных системных средств, так и на физическом уровне будут приведены в последующих статьях этой книги.

Статья 29. **Магнитные диски**

Программы и прочие документы, которые мы записываем на дискеты или жесткие диски, чтобы иметь возможность обращаться к ним в случае необходимости, хранятся там в виде файлов. Файл – это просто участок дискового пространства, содержащий тот или иной документ. Однако помимо файлов на диске имеется еще целый ряд специализированных системных областей, с помощью которых операционная система осуществляет управление файлами – поиск уже имеющихся файлов или создание новых, удаление (по команде пользователя) ненужных файлов и другие операции. Совокупность самих файлов и системных областей, предназначенных для их обслуживания, носит название файловой системы.

Наличие файловой системы избавляет пользователя от необходимости вникать в детали физической организации файлов на магнитном диске. В подавляющем большинстве случаев для работы с файлом требуется знать только его имя и каталог, в котором он находится. Иногда, однако, возникает необходимость глубже вникнуть в структуру файлов, в частности определить местонахождение файла на диске, рассмотреть его фактическое содержимое, возможно даже изменить значения отдельных байтов на диске. Такая задача может возникнуть при поиске файлов или системных областей, зараженных вирусами, при реализации процедуры защиты файлов от несанкционированного доступа и в других случаях. Так или иначе отчетливое представление о физической организации диска весьма полезно для серьезного программиста.

Хотя общие принципы разметки жестких дисков и дискет одинаковы, их физическая организация несколько различается, главным образом из-за значительно большей емкости жестких дисков и наличия на одном физическом жестком диске нескольких логических (C:, D:, E: и т. д.). Рассмотрим сначала организацию дискеты.

Современная дискета (для определенности рассмотрим наиболее широко используемый тип дискет диаметром 3,5 дюйма с емкостью 1,44 Мбайт) представляет собой гибкую пластину, на обе стороны которой информация записывается в виде магнитных меток, располагаемых по концентрическим дорожкам (рис. 29.1).

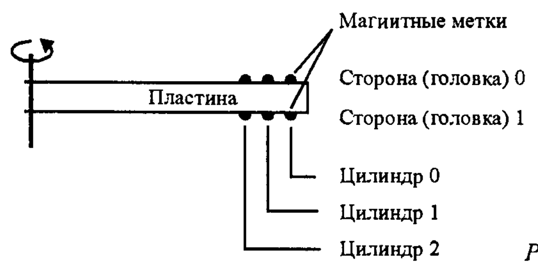


Рис. 29.1. Физическая организация дискеты

Дорожки обеих сторон пластины, расположенные на одном удалении от центра, объединяются понятием цилиндра. Цилиндры (как и дорожки) нумеруются от края диска внутрь, нумерация начинается с нуля. Стороны также нумеруются с нуля, и, поскольку каждую рабочую сторону диска обслуживает отдельная магнитная головка,

предназначенная для чтения и записи данных, часто говорят не о номерах сторон, а о номерах головок. Число дорожек определяет емкость дискеты. Дискеты емкостью 1,44 Мбайт содержат по 80 дорожек на каждой стороне, или, что то же самое, 80 цилиндров.

Каждая дорожка делится на секторы. Логический размер сектора для всех дисков составляет 512 байт. Число секторов на одной дорожке зависит от типа дискеты; дискеты емкостью 1,44 Мбайт имеют на каждой дорожке по 18 секторов (512 байт * 18 секторов * 80 цилиндров * 2 стороны = 1 474 560 байт = 1,44 Мбайт).

Контроллер диска позволяет читать или записывать информацию только целыми секторами. Невозможно прочитать с дискеты 1 байт; даже если мы заказываем в программе, написанной на каком-либо языке, чтение 1-байтовой переменной, физически с диска читается весь сектор, в котором находится эта переменная. Точно так же при физическом обращении к диску (т. е. при обращении не к конкретному файлу, а к некоторому месту на диске) следует указывать не номер байта, а номер сектора.

Существуют две системы нумерации секторов на диске – абсолютная (физическая) и относительная (логическая). Физическая нумерация секторов начинается с единицы (не с нуля!) на каждой дорожке, и каждая дорожка, таким образом, содержит секторы с номерами от 1 до 18. При использовании физической нумерации для указания конкретного сектора необходимо задать три параметра: номер цилиндра, номер стороны (головки) и номер сектора.

Относительная нумерация ведется в пределах всей дискеты. В этом случае секторы нумеруются с нуля; последний сектор имеет номер 2879. Для указания конкретного сектора достаточно задать его номер. При записи информации на диск DOS сначала целиком заполняет дорожки цилиндра 0, затем – цилиндра 1 и т. д., идя от периферии дискеты к ее центру. В таком же порядке возрастают логические номера секторов.

Полезно иметь в виду, что в действительности каждый сектор, кроме полезной информации, занимающей всегда точно 512 байт, имеет в своем составе еще информацию служебную, в частности свой собственный номер, чтобы его можно было найти, и контрольную сумму, с помощью которой контроллер может обнаружить возникающие на диске дефекты поверхности. Прикладному программисту практически никогда не приходится сталкиваться со служебной информацией, хранящейся в секторах диска, так как системные средства обращения к диску возвращают в программу только содержательную часть сектора.

Новая, пустая дискета в процессе форматирования, т. е. разбивки на секторы, получает следующие системные области:

- загрузочный сектор (Boot-сектор);
- таблицы размещения файлов – FAT (File Allocation Table). Для надежности на диске размещаются две FAT, дублирующие друг друга;
- корневой каталог.

Эти области располагаются в самом начале дискеты, занимая в общей сложности 33 сектора, или 16,5 Кбайт. Остальное пространство дискеты отводится под файлы и каталоги пользователя и иногда называется областью данных диска.

Загрузочный сектор (самый первый сектор дискеты с логическим номером 0) содержит физические характеристики дискеты в целом, а также программу начальной

загрузки операционной системы. Программа начальной загрузки фактически используется только в тех случаях, когда дискета является загрузочной, т. е. содержит основные программы операционной системы и служит для начальной загрузки компьютера. Однако присутствует эта программа на всех дискетах, в том числе и на тех, которые предназначены лишь для хранения данных. В табл. 29.1 приведен формат загрузочного сектора с указанием типичного для дискеты емкостью 1,44 Мбайт содержимого. Следует иметь в виду, что в зависимости от типа дискеты и того, какая программа использовалась для ее форматирования, содержимое отдельных полей загрузочного сектора может различаться.

Таблица 29.1. Формат загрузочного сектора

Смещение от начала сектора	*Число байтов*	*Назначение поля*	*Типичное содержимое для дискеты 1,44 Мбайт*
00h	3	Команда JMP перехода к программе начальной загрузки	EBh 3Ch 90h
03h	8	Сигнатура изготовителя операционной системы	'MSDOS5.0'
0Bh	2	Размер сектора в байтах	0200h=512
0Dh	1	Размер кластера в секторах	1
0Eh	2	Число секторов до первой FAT	1
10h	1	Число FAT	2
11h	2	Число записей в корневом каталоге	00E0h=224
13h	2	Полное число секторов на диске	0B40h=2880
15h	1	Байт описания носителя	F0h
16h	2	Размер FAT в секторах	0009h
18h	2	Число секторов на дорожке	12h=18
1Ah	2	Число головок (сторон)	2
1Ch	4	Число скрытых секторов	00000000h
20h	4	Число секторов на диске (для дисков больше 32 Мбайт)	00000000h
24h	1	80h=1-й жесткий диск	00h
25h	1	Зарезервировано	00h
26h	1	Сигнатура расширенного Boot-сектора (DOS 4.0 и больше)	29h
27h	4	Серийный номер (дается при форматировании)	–
2Bh	11	Метка тома (дается пользователем)	–
36h	8	Тип файловой системы	'FAT12 '
3Eh	450	Программа начальной загрузки и относящиеся к ней данные	–

При форматировании дискеты на ней резервируется место для корневого каталога. В дальнейшем по мере записи пользователем на дискету файлов и подкаталогов информация о них заносится в корневой каталог в виде 32-байтовых элементов. Поскольку размер корневого каталога жестко фиксирован, в нем помещается лишь огра-

ниченное количество элементов (224 для дискеты 1,44 Мбайт, см. табл. 29.1). Поэтому, если на дискете требуется разместить очень много небольших по размеру файлов, для них сначала следует создать хотя бы один подкаталог, а затем уже заполнять его файлами. Размеры подкаталогов, в отличие от корневого каталога, не имеют ограничений, и таким образом на дискету можно записать любое число файлов (разумеется, не больше, чем на нее поместится).

Формат элемента каталога (как корневого, так и любого вложенного) приведен на рис. 29.2.

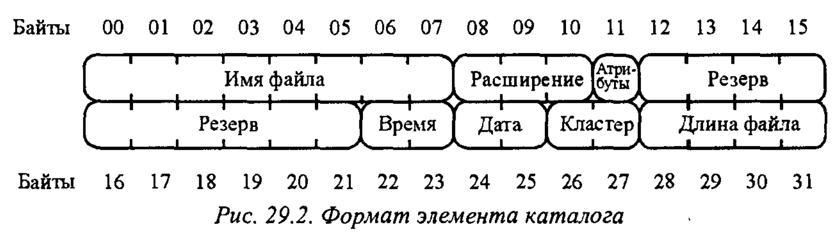


Рис. 29.2. Формат элемента каталога

В первых 11 байтах элемента каталога хранятся имя файла и его расширение. Обычно DOS при получении от пользователя имени файла автоматически преобразует символы имени и расширения, делая их прописными. Таким образом, в элементе каталога спецификация файла всегда указывается прописными буквами.

Атрибуты файла записываются независимо в отдельных битах байта атрибутов. Значения атрибутов приведены в табл. 29.2.

Таблица 29.2. Атрибуты файлов

Значение	*Сокращенное обозначение*	*Описание*
01h	R	Файл только для чтения
02h	H	Скрытый файл
04h	S	Системный файл
08h	L	Метка тома
10h	D	Каталог
20h	A	Файл не архивирован

Файл может иметь несколько атрибутов одновременно. Так, для записи о метке тома характерно значение байта атрибутов 28h: метка тома, не архивирована. Защищенный от стирания и модификации файл содержит в байте атрибутов код 21h, а системным файлам IO.SYS и MSDOS.SYS с помощью кода 7h назначается комбинация атрибутов: только для чтения, скрытый и системный.

Время и дата создания или последней модификации файла записывается в виде двоичных чисел в отдельных полях отведенных под эту информацию слов, как это показано на рис. 29.3.

В байтах 26...27 элемента каталога записывается номер кластера, с которого начинается файл на диске. Хотя минимальной порцией информации, передаваемой контроллером диска в процессе записи или чтения файла, является сектор, файловая система DOS назначает место на диске целыми кластерами. Минимальный физический размер файла, даже если полезные данные в нем занимают лишь несколько байтов, составляет один кластер.

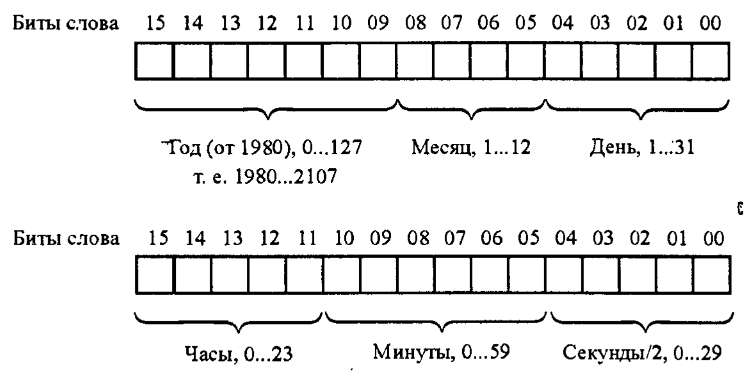


Рис. 29.3. Форматы записи даты и времени в каталоге диска

Размер кластера на рассматриваемой дискете совпадает с размером сектора (512 байт), но для дисков другого типа это не так. Например, кластер дискеты 5,25 дюйма емкостью 360 Кбайт содержит два сектора; кластеры жестких дисков обычно включают 4…8 секторов. Кластеры существуют только в области данных диска (не в его системных областях) и нумеруются последовательно, причем считается, что номер первого кластера равен двум. Таким образом, для дискеты рассматриваемого типа кластер номер 2, характеризующий самое начало области данных диска, совпадает с относительным сектором 33. Файл не обязательно должен быть слитным, его отдельные кластеры могут быть разбросаны по всему диску. Поэтому для описания размещения файла на диске следует где-то хранить номера всех входящих в него кластеров. Эта информация и составляет содержимое таблицы размещения файлов FAT. В элементе каталога находится только номер первого кластера файла.

Таблица размещения файлов FAT является важнейшей структурой диска. По ее содержимому можно определить физическое расположение на диске всех файлов и каталогов. Таблица размещается в секторах 1...9, а ее дубликат – в секторах 10...17 дискеты. Таблица начинается с байта описания носителя (того самого, который хранится в байте 15h загрузочного сектора), за которым следуют 2 неиспользуемых байта. Таким образом, информация о файлах начинается с байта 3. Все пространство FAT разбито на поля по 1,5 байта (3 полубайта), которые закрепляются за последовательными кластерами диска.

Таким образом, первое 1,5-байтовое поле FAT, начинающееся с байта 3 этой таблицы, соответствует кластеру 2; следующее 1,5-байтовое поле – кластеру 3 и т. д. Как уже отмечалось, в элементе каталога хранится номер первого кластера файла. В поле FAT, соответствующем этому кластеру, записывается номер второго кластера этого файла; в поле FAT, соответствующем второму кластеру файла, записывается номер третьего кластера этого файла и т. д. Поле кластера, в котором файл заканчивается, помечается кодом FFFh (табл. 29.3).

Таблица 29.3. Возможные значения полей FAT

Код	*Описание*
000h	Кластер свободен
FFFh	Последний кластер файла
*nnn*h	Номер следующего кластера файла
FF7h	Дефектный кластер

В таблице размещения файлов нет сведений ни об именах файлов, ни об их начальных адресах – эти данные хранятся в элементах каталогов, соответствующих конкретным файлам. В FAT записана лишь информация о продолжении каждого файла – какие кластеры он занимает и где кончается. Если, например, в кластерах 2, 3 и 4 расположены три относительно коротких файла, а следующий файл занимает кластеры 5, 6, 7 и 8, то последовательные поля FAT (от поля, закрепленного за кластером 2, до поля, соответствующего кластеру 8) будут заполнены следующими кодами:

```
FFF  FFF  FFF  006  007  008  FFF
```

Следует иметь в виду, что при выводе на экран терминала содержимого FAT (по байтам) эти данные будут выглядеть по-другому:

```
FF  FF  FF  FF  0F  60  00  07  80  FF  xF
```

Это происходит потому, что при выводе на экран числа, хранящегося в целом байте (два полубайта), вначале выводится старшая цифра, а затем младшая. Поэтому последовательные полубайты с номерами 0, 1, 2, ... будут выведены на экран в таком порядке:

```
1,0  3,2  5,4  7,6  ...
```

Повреждение FAT влечет за собой полную потерю информации о файлах на диске. Хотя сами файлы могли остаться неповрежденными, однако теряется возможность узнать, где находятся их продолжения (для файлов, длина которых превышает один кластер). Именно это обстоятельство побудило разработчиков файловой системы предусмотреть на каждом диске два экземпляра FAT, которые при нормальных обстоятельствах полностью дублируют друг друга. Часто бывает, что в силу какого-либо программного сбоя одна из копий FAT повреждается; тогда с помощью специальных инструментальных средств (например, программы SCANDISK) можно скопировать на место поврежденной копии ее неповрежденный дубликат и восстановить работоспособность диска.

Приведенные сведения о логической организации диска позволяют пояснить механизм удаления (и восстановления) файлов. Возможность восстановления ошибочно удаленных файлов основана на том, что при удалении файла (например, командой DOS DEL) сам файл физически не стирается, а остается на диске в точности в том же виде, каким он был до удаления. В чем же тогда заключается удаление файла? При удалении файла DOS выполняет две операции: заменяет первую букву имени файла в записи каталога кодом E5h, соответствующим (на русифицированном компьютере) строчной букве "x", а в обеих копиях таблицы размещения файлов помечает свободными все кластеры, принадлежащие файлу. Таким образом, для удаленного файла сохраняется большая часть его имени и все расширение, а также его длина и номер начального кластера. С другой стороны, теряется информация о номерах последующих кластеров файла. Однако очень часто файл либо оказывается слитным, занимая на диске последовательные кластеры, либо заполняет промежутки между другими файлами. Специальные программы восстановления позволяют проанализировать дисковое пространство и, сделав предположение о возможном местонахождении удаленного файла, восстановить его, дописав первую букву имени и заполнив соответствующим образом обе копии FAT. Однако полной уверенности в правильности восстановления случайно удаленного файла не может быть никогда, за исключением тех случаев, когда файл занимает лишь один кластер. Разумеется, возможность восстановления файла

полностью теряется, если созданные позднее файлы физически затрут его на диске или если окажется затертой относящаяся к нему запись каталога.

В случае необходимости прочитать информацию из системных областей дискеты можно воспользоваться прерыванием DOS 25h, которое позволяет обратиться к любому сектору диска по его логическому номеру. Поскольку на дискете все секторы имеют логические номера (как будет показано ниже, для жесткого диска это не так), с помощью прерывания 25h можно прочитать любой сектор дискеты (для записи используется прерывание 26h). В примере 29.1 читается первый сектор корневого каталога дискеты.

Пример 29.1. Чтение первого сектора корневого каталога

```
mov   AX,data       ;Инициализация
mov   DS,AX         ;регистра DS
mov   AL,0          ;Код диска A:
mov   BX,offset buf;Смещение буфера в программе
mov   CX,1          ;Число читаемых секторов
mov   DX,19         ;Номер сектора
int   25h           ;Вызов DOS
pop   AX            ;Восстановление стека
```

В полях данных программы выделяется буфер buf объемом 512 байт, куда поступит содержимое читаемого сектора. Прерывание 25h требует для своего выполнения целого ряда параметров. В регистр AL заносится код читаемого диска (0 – A:, 1 – B:, 2 – C: и т. д.). В регистрах DS:BX должен быть адрес буфера для записи возвращаемой информации. В регистр CX заносится число читаемых секторов (за один раз можно прочитать более одного сектора), а в регистр DX – номер первого сектора, подлежащего чтению. Прерывание 25h оставляет смещенным стек, и для его восстановления после возврата из DOS необходимо выполнить одну команду pop. В результате выполнения приведенного фрагмента в буфер buf будет прочитано двоичное содержимое первого сектора корневого каталога дискеты, установленной на дисководе A:. Не составляет труда сохранить содержимое буфера buf в файле, а затем вывести на экран в виде 16-ричного дампа (например, с помощью встроенного редактора программы Norton Commander), как это сделано на рис. 29.4 для конкретной дискеты. Видно, что на дискету были записана метка тома DISKETTE 38 (она фигурирует в корневом каталоге в виде записи о файле с атрибутом 28h), а также три коротких (по 5 байт) файла с именами FILE1.DAT, FILE2.DAT и FILE3.DAT. Эти файлы заняли начальные кластеры дискеты с номерами 2, 3 и 4. Воспользовавшись рис. 29.2 и 29.3, можно расшифровать поля записи каталога с датой и временем создания файлов.

```
00000  44 49 53 4B | 45 54 54 45 | 20 33 38 28 | 00 00 00 00   DISKETTE 38(....
00010  00 00 00 00 | 00 00 DD A0 | 24 29 00 00 | 00 00 00 00   ......▌a$)......
00020  46 49 4C 45 | 31 20 20 20 | 44 41 54 20 | 00 00 00 00   FILE1   DAT ....
00030  00 00 00 00 | 00 00 F4 A0 | 24 29 02 00 | 05 00 00 00   ......Їa$)☻.♣...
00040  46 49 4C 45 | 32 20 20 20 | 44 41 54 20 | 00 00 00 00   FILE2   DAT ....
00050  00 00 00 00 | 00 00 00 A1 | 24 29 03 00 | 05 00 00 00   .......б$)♥.♣...
00060  46 49 4C 45 | 33 20 20 20 | 44 41 54 20 | 00 00 00 00   FILE3   DAT ....
00070  00 00 00 00 | 00 00 07 A1 | 24 29 04 00 | 05 00 00 00   .......б$)♦.♣...
00080  00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00   ................
00090  00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00   ................
000A0  00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00   ................
000B0  00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00   ................
```

Рис. 29.4. Корневой каталог дискеты

Жесткий диск отличается от дискеты прежде всего наличием не одной, а нескольких пластин (рис. 29.5), а также значительно большим числом секторов на каждой дорожке и дорожек на каждой поверхности, что позволяет существенно повысить емкость магнитного диска.

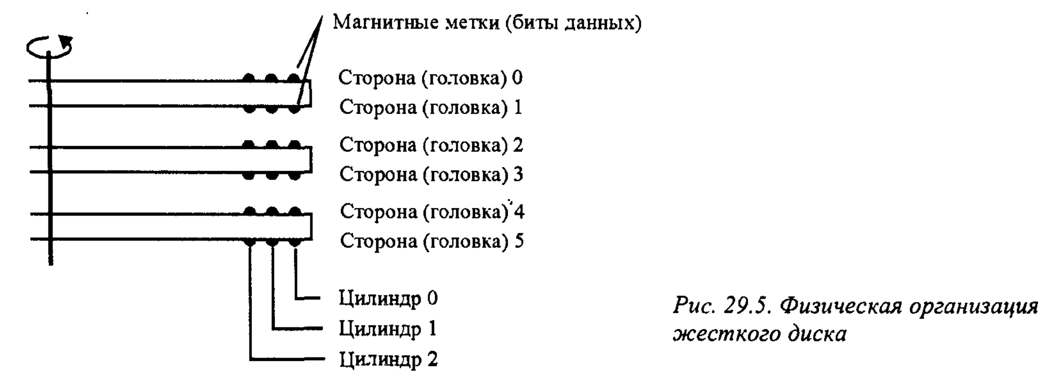


Рис. 29.5. Физическая организация жесткого диска

Самый первый сектор жесткого диска (сектор 1 стороны 0 цилиндра 0) называется главным загрузочным сектором (Master boot); он включает в себя программу главного загрузчика и таблицу разделов диска (рис. 29.6). В процессе начальной загрузки компьютера программы BIOS загружают в оперативную память главный загрузочный сектор и передают управление на программу главного загрузчика. Эта программа анализирует таблицу разделов и отыскивает в ней раздел, являющийся загрузочным (как правило, логический диск С:). В память загружается загрузочный сектор этого диска, и управление передается на программу начального загрузчика.

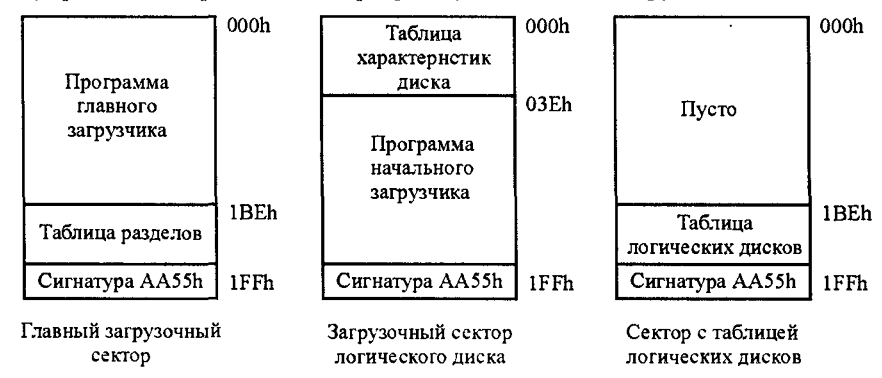


Рис. 29.6. Структуры служебных полей жесткого диска

В таблице разделов указываются адреса и размеры разделов, на которые разбит жесткий диск. Всего в таблице разделов зарезервировано место для четырех 16-байтовых записей о разделах, которых, таким образом, не может быть больше четырех. Однако DOS предоставляет возможность создания не более двух разделов. Один из них, называемый первичным, является загружаемым диском С:. Второй раздел называется расширенным; в нем можно создать один или несколько логических дисков (D:, E: и т. д. до Z:). Формат записи таблицы разделов приведен в табл. 29.4.

Таблица 29.4. Запись таблицы разделов жесткого диска

Смещение от начала записи	*Число байтов*	*Назначение*
00h	1	Флаг загружаемого раздела; 80h – раздел загружаемый, 00h – не загружаемый
01h	1	Начальная головка
02h	2	Начальные цилиндр и сектор
04h	1	Тип файловой системы
05h	1	Конечная головка
06h	2	Конечные цилиндр и сектор
08h	4	Начальный относительный сектор
0Ch	4	Число секторов в разделе

Номера цилиндра и сектора (как начальных, так и конечных) упакованы в одно слово, в котором номер сектора занимает биты 5...0, а номер цилиндра записан "вперемежку": старшие 2 бита номера цилиндра хранятся в битах 7 и 6 слова, а остальные 8 бит – в старшем байте слова, т. е. в битах 15...8.

Возможные значения типа файловой системы приведены в табл. 29.5.

Таблица 29.5. Коды типа файловой системы раздела жесткого диска

Код	*Тип файловой системы*
00h	Неизвестный тип
01h	12-битовая FAT; раздел меньше 10 Мбайт
04h	16-битовая FAT; раздел меньше 32 Мбайт
05h	Расширенный раздел DOS
06h	16-битовая FAT; раздел равен или больше 32 Мбайт

Таблица размещения файлов FAT на дискетах и жестких дисках небольшого объема состоит из полей размером 1,5 байта (12 бит), что не позволяет иметь на диске более 4096 кластеров. На дисках большего объема используется 16-битовая FAT, в которой поля для записи кластеров имеют размер 2 байта. Это увеличивает размер FAT, но позволяет располагать на диске до 65536 кластеров. Обычно на эти два типа файловой системы ссылаются как на FAT12 и FAT16.

Разметка диска, т. е. разбиение его на разделы, осуществляется с помощью программы DOS FDISK. Обычно на физическом диске создают два раздела: первичный (тип 06h) и расширенный (тип 05h), в котором организуют 2, 3 или более логических диска, количество которых выбирают исходя из полного объема физического диска и условий использования компьютера.

Так или иначе жесткий диск оказывается разбит на логические диски, каждый из которых занимает целое число цилиндров; диск C: начинается с начала первой свободной дорожки после главного загрузчика, т. е. с сектора 1 стороны 1 цилиндра 0. Каждому логическому диску, входящему в расширенный раздел, предшествует сектор, содержащий таблицу логических дисков (см. рис. 29.6). В этой таблице указываются адреса начала и конца данного логического диска. Таким образом DOS, просматривая последовательно таблицы логических дисков, может постепенно "добраться" до адреса любого логического диска расширенного раздела.

Таблица логических дисков располагается в самом первом секторе области, выделенной под логический диск, т. е. в секторе 1 стороны 0 цилиндра n. Оставшаяся часть этой дорожки не используется; таким образом, собственно логический диск начинается в секторе 1 стороны 1 того же цилиндра. Каждый диск начинается со своего загрузочного сектора (Boot), включающего таблицу характеристик диска и программу начальной загрузки (используемую только на загрузочном диске C:). Вслед за загрузочным сектором располагаются две таблицы размещения файлов (FAT) и корневой каталог. Все остальное пространство логического диска отводится под область каталогов и файлов (рис. 29.7).

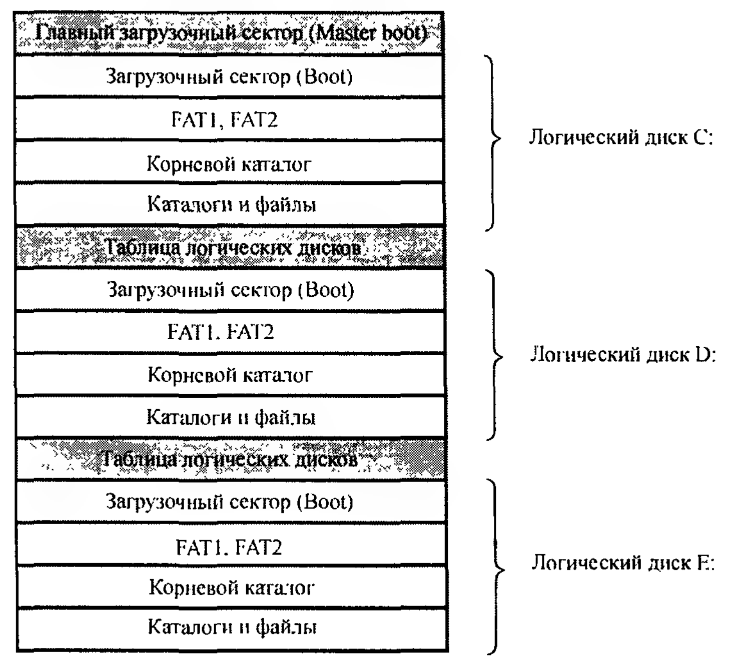


Рис. 29.7. Структура физического диска, разбитого на логические диски

Каждый логический диск имеет свою относительную нумерацию секторов, начинающуюся с нуля. Таким образом, относительный номер сектора надо использовать с указанием логического диска, при этом относительный сектор 0 соответствует загрузочному сектору, относительный сектор 1 – первому сектору первой копии FAT и т. д. Сектор с таблицей логических дисков (как и сектор главного загрузчика) не входит в систему относительной нумерации секторов и не принадлежит логическому диску. Между прочим, именно по этой причине инструментальные пакеты, работающие не со всем физическим диском, а с логическими дисками (например, PC Tools), не позволяют прочитать или модифицировать главный загрузчик и таблицы логических дисков.

Для вычисления номера сектора, относящегося к какому-то файлу или системной области диска, необходимо иметь такие характеристики диска, как размеры сектора и кластера, число секторов, составляющих FAT и корневой каталог, размеры и адреса логических дисков, номера кластеров, отводимых под искомый файл. Эти данные извлекаются из загрузочных секторов дисков, корневых и вложенных каталогов, а также

таблиц разделов, логических дисков и размещения файлов. При исследовании диска "вручную" удобно воспользоваться программой DiskEdit из пакета нортоновских утилит, которая позволяет обратиться ко всем областям физического диска, или программой PC Tools, с помощью которой можно изучать только содержимое логических дисков. Если же поиск, чтение или модификацию областей диска необходимо выполнять программно, то для обращения к главному загрузчику и таблицам логических дисков следует воспользоваться прерыванием BIOS 13h; секторы, входящие в логические диски (загрузочные секторы, FAT, каталоги и файлы), можно читать и модифицировать с помощью того же прерывания BIOS 13h, однако проще воспользоваться прерываниями DOS 25h и 26h (см. пример 29.1).

Некоторое представление о методике работы с системными областями жесткого диска может дать приводимый ниже фрагмент программы (пример 29.2), в которой выполняется чтение первой таблицы логических дисков (описывающей логический диск D:, см. рис. 29.7). В этом примере с помощью прерывания BIOS 13h читается сектор главного загрузчика (головка 0, цилиндр 0, сектор 1) и из содержащейся в нем таблицы разделов извлекается номер последнего цилиндра первичного раздела диска (т. е. логического диска C:). Добавление единицы к номеру последнего цилиндра диска C: даст номер цилиндра, на котором находится первая таблица логических дисков (в секторе 1 стороны 0 этого цилиндра). Сектор с искомой таблицей читается вторым вызовом прерывания 13h.

Пример 29.2. Чтение первой таблицы логических дисков

```
;В сегменте команд
main    proc                ;(1)
        mov   AX,data       ;(2)Настроим регистр ES
        mov   ES,AX         ;(3)на сегмент данных
;Прочитаем Master boot
        mov   AH,02h        ;(4)Функция чтения секторов
        mov   AL,1          ;(5)Число читаемых секторов (1)
        mov   BX,offset buf;(6)ES:BX=адрес буфера
        mov   CH,0          ;(7)Цилиндр (0)
        mov   CL,1          ;(8)Сектор (1) и старшие биты номера цилиндра
        mov   DH,0          ;(9)Головка (0)
        mov   DL,80h        ;(10)Физический диск (жесткий)
        int   13h           ;(11)Вызов BIOS
;Выделим номер последнего цилиндра первичного раздела
        mov   AX,word ptr buf+1BEh+6;(12)Последний цилиндр раздела
        mov   byte ptr cyl,AH;(13)Младший байт номера цилиндра
        mov   CL,6          ;(14)Будет сдвиг на 6 бит
        shr   AL,CL         ;(15)Биты 6 и 7 в начало байта
        mov   byte ptr cyl+1,AL;(16)Добавим их к номеру цилиндра
        inc   cyl           ;(17)Получим номер следующего цилиндра
; Прочитаем таблицу логических дисков
        mov   AH,02h        ;(18)Функция чтения секторов
        mov   AL,1          ;(19)Число читаемых секторов
        mov   BX,offset buf;(20)ES:BX=адрес буфера
        mov   DH,byte ptr cyl;(21)Младший байт цилиндра в DH
        mov   DL,byte ptr cyl+1;(22)Старший байт цилиндра в DL
        mov   CL,6          ;(23)Сдвинем 2 младших бита
        shl   DL,CL         ;(24)в разряды 6,7
        or    DL,1          ;(25)Добавим номер сектора
        mov   CX,DX         ;(26)Перенесем весь параметр в CX
        mov   DH,0          ;(27)Головка
        mov   DL,80h        ;(28)Жесткий диск
        int   13h           ;(29)Вызов BIOS
```

```
main    endp                ;(30)
;В сегменте данных
buf     db 512 dup (?)      ;(31)Буфер для чтения
cyl     dw    0             ;(32)Временное хранение номера цилиндра
```

Функция 02h прерывания 13h требует для своего выполнения передачи ей через регистры общего назначения следующих параметров:

- AH=02h (номер функции);
- AL=число читаемых секторов (у нас один);
- CH=младший байт номера цилиндра (8 бит);
- CL=начальный читаемый сектор (биты 0...5) и 2 старших бита номера цилиндра (биты 6 и 7 в CL);
- DH=головка;
- DL=дисковод (0 или 1 – гибкие диски, 80h – первый жесткий диск);
- ES:BX=адрес буфера, в который поступает прочитанная функцией информация.

В предложениях 2 и 3 сегментный регистр ES настраивается на сегментный адрес (data) сегмента данных. Эта операция необходима из-за того, что функция 02h прерывания 13h возвращает прочитанный сектор по адресу, указываемому в паре регистров ES:BX. Далее в регистры заносятся необходимые параметры и вызовом BIOS (предложение 11) выполняется чтение сектора главного загрузчика. Как уже отмечалось выше, начало сектора занято собственно программой начальной загрузки, а начиная с байта 1Bh располагается таблица разделов. Номер последнего цилиндра раздела (вместе с номером сектора) находится в слове со смещением 6 относительно начала этой таблицы (см. табл. 29.4 и текст к ней). Это слово переносится в регистр AX для последующей обработки (предложение 12). Заметим, что формат этого слова совпадает с форматом регистра CX при вызове функции 02h прерывания 13h: биты 0...5 занимает номер сектора, биты 6 и 7 – старшие 2 бита номера цилиндра, а биты 8...15 (соответствующие регистру CH) – младшие 8 бит номера цилиндра. Если бы нам нужно было прочитать последний сектор последнего цилиндра диска C:, слово со смещением 6 из таблицы разделов можно было бы просто перенести в регистр CX. Однако мы хотим прочитать первый сектор следующего цилиндра. Для получения его номера нам придется преобразовать слово со смещением 6, выделить из него номер цилиндра (предложения 12...15), прибавить к нему 1 (предложение 17), а при формировании параметров второго вызова прерывания 13h выполнить обратную операцию слияния в одном слове номеров цилиндра и сектора (предложения 21...25). Эти преобразования сначала осуществляются в регистре DX (регистр CL нужен для выполнения команды shl), а затем сформированный параметр переносится в регистр CX (предложение 26).

В результате выполнения приведенного выше фрагмента в буфер buf будет прочитан весь сектор с первой таблицей логических дисков. Участок этого буфера, начинающийся со смещения 1BEh, представляет собой искомую таблицу разделов. На рис. 29.8 приведен 16-ричный дамп последних 80 байт сектора с таблицей логических дисков конкретного компьютера. Анализируя содержимое таблицы, можно увидеть, что логический диск D: начинается с цилиндра с номером 5Eh=94, а заканчивается цилиндром с номером 9Eh=158. Таким образом, объем диска D: составляет 64 цилиндра, или приблизительно 500 Мбайт (рассматриваемый диск имел 255 головок при 63 секторах на каждой дорожке).

```
001B0  00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 01   ...............☺
001C0  01 5E 06 FE | 3F 9D 3F 00 | 00 00 01 B0 | 0F 00 00 00   ☺^♠■?Э?...☺░☼...
001D0  01 9E 05 FE | 3F DD 40 B0 | 0F 00 40 B0 | 0F 00 00 00   ☺Ю♣■?▌@░☼.@░☼...
001E0  00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00   ................
001F0  00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 55 AA   ..............Uк
```

Рис. 29.8. Первая таблица логических дисков

Статья 30. **Параллельный интерфейс**

Как уже отмечалось ранее (см. статью 23), основным средством связи между различными устройствами, входящими в состав компьютера, является системная магистраль. Устройства подключаются к магистрали через управляющие блоки, обычно называемые контроллерами или адаптерами. Именно так организуется работа, например, видеосистемы, магнитных дисков, клавиатуры. Во многих случаях так же осуществляется и расширение состава компьютера: новые устройства (точнее, их управляющие блоки, выполняемые в виде плоских печатных плат) подключаются к системной магистрали через предусмотренные для этого резервные разъемы системной платы – "слоты". Этот способ используется, например, для включения в состав компьютера звуковых плат или некоторых модемов.

Имеется и другой способ комплектации компьютера дополнительным внешним оборудованием. Он заключается в использовании последовательного и параллельного интерфейсов – стандартных средств передачи данных, адаптеры которых всегда входят в состав современных компьютеров. Эти адаптеры подключены к системной магистрали, что обеспечивает возможность их программного управления, а их выходы оформлены в виде разъемов, выведенных на заднюю панель компьютера. Как правило, компьютер комплектуется одним выводом параллельного интерфейса и двумя независимыми выводами последовательного, хотя и тех и других выводов может быть больше. В стандартной конфигурации один из разъемов последовательного интерфейса используется для подключения мыши (другой можно использовать для связи с другими компьютерами или для подключения внешнего модема), а через параллельный интерфейс осуществляется подключение принтера. И тем и другим интерфейсом можно воспользоваться для подключения нестандартного (например, измерительного) оборудования. Рассмотрим сначала особенности функционирования параллельного интерфейса.

В компьютерах используется разновидность параллельного интерфейса под названием Centronics, отличающаяся относительно высокой скоростью передачи данных (до 150 Кбайт/с) и простотой программирования. Centronics позволяет передавать данные только в одном направлении – из компьютера в устройство, однако при необходимости передать данные из нестандартного устройства в компьютер это можно сделать с помощью четырех линий состояния интерфейса. Впрочем, используется и разновидность параллельного интерфейса с возможностью двухсторонней передачи данных. Разумеется, в установке, подключаемой к компьютеру через параллельный интерфейс, должно быть предусмотрено устройство сопряжения, воспринимающее и вырабатывающее сигналы обмена с интерфейсом.

Интерфейс Centronics подключается к периферийному устройству (принтеру) с помощью кабеля, содержащего 17 сигнальных линий и несколько линий нуля. Управление интерфейсом осуществляется через три закрепленных за ним порта: порта данных с адресом 378h, порта состояния принтера с адресом 379h и порта управления принте-

ром с адресом 37Ah. Порты фактически представляют собой 8-разрядные регистры, биты которых соответствуют сигналам интерфейса. Некоторые из этих сигналов (конкретно – сигналы портов данных и управления) являются для интерфейса выходными; их должна устанавливать программа, управляющая передачей информации. Другие сигналы, наоборот, поступают из периферийного устройства и отображаются в состоянии закрепленных за ними битов порта состояния; программа должна читать и анализировать эти биты.

На рис. 30.1 показаны порты интерфейса Centronics с указанием сигналов, соответствующих конкретным битам (апостроф (') после названия сигнала обозначает, что на данной линии действует сигнал обратной полярности).

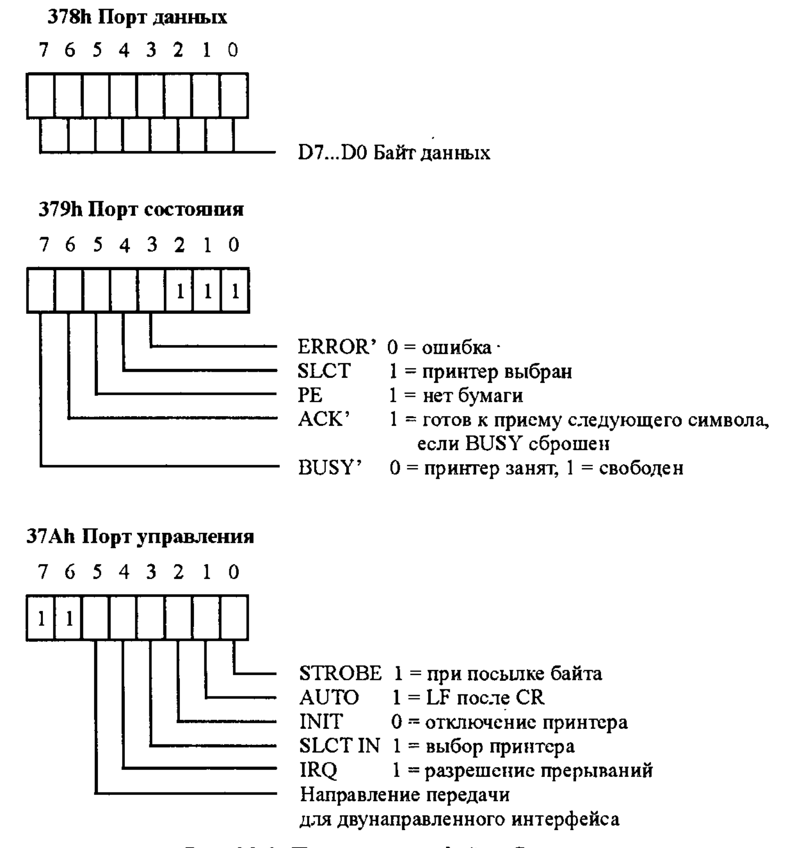


Рис. 30.1. Порты интерфейса Centronics

Программирование параллельного интерфейса требует некоторых сведений о его протоколе, т. е. последовательности и взаимодействии сигналов, которыми интерфейс обменивается с подключенным к нему устройством. Некоторые из этих сигналов имеют узкоспециализированное назначение и возникают лишь в особых случаях (например, сигнал PE – конец бумаги), другие же принимают обязательное участие в процедуре передачи данных. К последним относятся 8 бит данных и три управляющих сиг-

нала STROBE', BUSY и ACK' (рис. 30.2). (Апостроф обозначает то же, что и на рис. 30.1.)

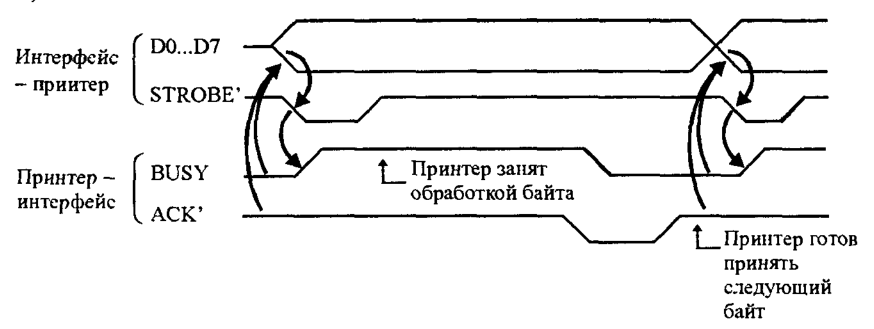


Рис. 30.2. Протокол передачи данных для интерфейса Centronics

Сигнал BUSY считается активным, когда он имеет высокое значение. В противоположность этому активное состояние сигналов STROBE' и ACK' низкое, отчего они и обозначаются с тем или иным дополнительным значком (с чертой наверху, со знаком минус или с апострофом, как у нас). Прослеживая соответствие сигналов интерфейса состоянию битов его портов, необходимо иметь в виду, что для некоторых сигналов (SLCT, PE, STROBE) в порты записываются их прямые значения, а для других (ERROR, ACK, BUSY) – инверсные.

Вывод на принтер каждого байта данных состоит из трех этапов. Прежде всего программа должна дождаться неактивного состояния сигналов BUSY и ACK (это и есть ожидание готовности устройства). Убедившись, что биты 6 и 7 порта состояния 379h установлены в 1 (см. рис. 30.1), программа посылает в порт данных 378h байт данных, что приводит к установке кода данных на линиях интерфейса D7...D0. Наконец, программа должна установить на короткое время сигнал STROBE, что реализуется путем установки и затем сброса бита 0 порта управления 37Ah. Следующие байты посылаются точно таким же образом.

Выполняя все эти операции, необходимо учитывать временные характеристики интерфейса. Сигнал STROBE можно посылать в порт управления не ранее чем через 0,5 мкс после установки данных, что может потребовать введения в программу небольшой программной задержки (одной или нескольких команд jmp, см. приведенный ниже текст программы). То же относится и к длительности сигнала STROBE, которая не должна быть меньше той же величины 0,5 мкс. Практически программные задержки часто оказываются не нужны.

Обратимся еще раз к рис. 30.2. Принтер, сняв с линий данных байт данных и начав его обработку (вывод на печать или сохранение во внутренней памяти), устанавливает ответный сигнал BUSY, действующий все время, пока принтер занят обработкой байта данных. Закончив обработку байта, принтер на некоторое время устанавливает сигнал ACK и сбрасывает сигнал BUSY. Окончание сигнала ACK (при сброшенном состоянии сигнала BUSY) говорит интерфейсу об окончании данной операции обмена и о возможности посылки следующего байта данных. Ввиду краткости сигнала ACK часто оказывается, что ожидать его снятия нет необходимости; достаточно дождаться неактивного состояния сигнала BUSY (т. е. 1 в бите 7 порта состояния). Вообще следует заметить, что различные принтеры могут

несколько по-разному выполнять свою часть протокола обмена. Рассмотренный ниже пример отлаживался на принтере Epson LQ-100.

Приведем текст программы (пример 30.1), в которой принтер программируется, как говорят, на физическом уровне, т. е. путем обращения к его портам. Разумеется, в большинстве случаев для вывода на принтер текста из выполняемой программы проще воспользоваться функциями DOS. Однако в некоторых специальных случаях приходится прибегать и к программированию через порты, например если принтер используется в нестандартном режиме или параллельный интерфейс служит для связи с нестандартным устройством.

Пример 30.1. Программирование параллельного интерфейса

```
        mov   CX,10      ;Повторить 10 раз
        mov   DX,379h    ;Порт состояния
wait1:  in    AL,DX      ;Чтение состояния
        and   AL,0C0h    ;Оставляем биты 7(BUSY) и 6(ACK), маскируем бит 4(SLCT)
        cmp   AL,0C0h    ;BUSY=ACK=1?
        jne   wait1      ;Нет, продолжать опрос порта
sym:    mov   AL,'*'     ;Символ для печати
        mov   DX,378h    ;Порт данных
        out   DX,AL      ;Вывод символа
        mov   DX,37Ah    ;Порт управления
        in    AL,DX      ;Читаем из порта. Там CCh (SLCT IN =1, INIT=1)
        jmp   $+2        ;Небольшая задержка
        or    AL,1       ;Устанавливаем сигнал STROBE
        out   DX,AL      ;В порт
        jmp   $+2        ;Небольшая задержка
        and   AL,0FEh    ;Сбрасываем сигнал STROBE
        out   DX,AL      ;В порт
        loop  sym        ;Цикл
```

В приведенном примере предполагается, что принтер выбран и установлен в исходное рабочее состояние, что обычно выполняется автоматически при его включении. Свидетельством этого будут установленные биты 2 и 3 (SLCT IN и INIT) в порте управления, а также бит 4 (SLCT) в порте состояния. В программе не выполняется анализ байта состояния на наличие ошибки или конца бумаги, что при работе с принтером, вообще говоря, следует предусматривать.

Статья 31. **Последовательный интерфейс**

Рассмотренный выше параллельный интерфейс отличается тем, что передача данных между компьютером и внешним устройством осуществляется по 8-разрядному каналу сразу целым байтом. Такая параллельная передача данных обеспечивает относительно высокую скорость обмены информацией, но требует многопроводных каналов связи (напомним, что разъем Centronics содержит 17 сигнальных линий). В тех случаях, когда источник или приемник информации удален от компьютера на значительное расстояние, а также когда нежелательно использовать многопроводной кабель, применяется последовательная передача, при которой данные проходят по единственной линии связи последовательно бит за битом. Это позволяет существенно упростить соединение, но, с другой стороны, заметно (по меньшей мере на порядок) уменьшает скорость передачи данных.

Различают два способа последовательной передачи данных: синхронный и асинхронный. В первом случае данные передаются сплошным потоком, без разделения на байты. Для идентификации отдельных битов источник и приемник должны управляться синхронно работающими генераторами. Во втором случае начало и конец каждой порции информации (кадра) отмечаются специальными метками; требования к синхронизации передатчика и приемника заметно снижаются, хотя падает и скорость передачи (за счет меток начала и конца кадра). Последовательный интерфейс компьютера использует асинхронную передачу.

Стандартный формат последовательной асинхронной передачи данных изображен на рис. 31.1. Уровень логической единицы в линии называется маркером, уровень логического нуля – пробелом. При отсутствии данных в линии действует сигнал маркера. Передача порции данных начинается с посылки стартового бита пробела. После этого передаются биты данных, число которых в кадре может устанавливаться от 5 до 8, хотя реально кадр всегда соответствует байту. За битами данных может следовать бит контроля четности (паритета). Этот бит выбирается в каждой порции данных таким образом, чтобы общее число единиц в битах данных и бите паритета было четным (или нечетным). Кадр заканчивается стоп-битом, имеющим уровень маркера. После этого в линии может поддерживаться состояние отсутствия данных (уровень маркера) или начинаться следующий кадр (стартовым битом пробела).

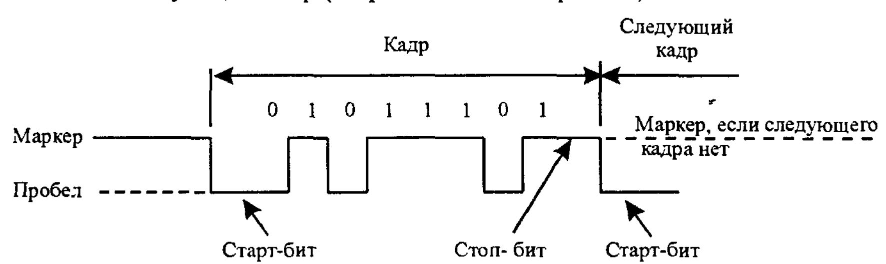


Рис. 31.1. Структура кадра при последовательной передаче данных

Сигналы в линии могут иметь различное представление. При передаче на небольшие расстояния в линии действуют биполярные уровни напряжения –15В/+15 В (этот способ передачи носит название стандарта RS-232). При больших расстояниях (до 1,5 км) используют токовую петлю – импульсы постоянного тока значением 20 или 40 мА, передаваемые по двухпроводной линии. Наконец, при передаче информации на неограниченные расстояния по телефонным линиям уровни напряжения преобразуют в посылки (пачки) синусоидальных сигналов. Сигналу маркера соответствует более высокая частота (тон), сигналу пробела – более низкая. Задачу преобразования сигналов напряжения, действующих на выходе последовательного интерфейса, в сигналы телефонного тона выполняет специальное устройство – модем, часто входящее в состав оборудования современного компьютера.

Обычно компьютер имеет два встроенных последовательных интерфейса (два последовательных порта), называемых COM1 и COM2. Выходные разъемы этих интерфейсов включают целый ряд сигналов, входящих в протокол RS-232 и обеспечивающих полное управление модемом (рис. 31.2).

Наименование	Номера контактов		Направление (ПК ↔ аппаратура)	Функция
	25-контактный	9-контактный		
TD	2	3	→	Передаваемые данные
RD	3	2	←	Принимаемые данные
RTS	4	7	→	Запрос передачи
CTS	5	8	←	Готов к передаче
DTR	20	4	→	ПК готов
DSR	6	6	←	Аппаратура готова
RCD	8	1	←	Детектор принимаемого сигнала
RI	22	9	←	Индикатор вызова
FG	1	–		Защитное заземление (корпус)
SG	7	5		Сигнальное заземление

Рис. 31.2. Сигналы интерфейса RS-232 и их разводка по контактам разъемов

Помимо выходной и входной линий, через которые осуществляется собственно передача данных, в протоколе RS-232 предусмотрены сигналы запроса и готовности передачи, запроса и готовности принимающей аппаратуры и ряд других. В зависимости от вида аппаратуры, подключаемой к компьютеру через последовательный порт, используется та или иная часть этих сигналов. В простейшем случае связь осуществляется по трехпроводному кабелю, две линии которого служат для передачи данных в одну и другую сторону, а третья является нулевой.

Порт COM1 обычно используется для подсоединения мыши; через порт COM2 к компьютеру подключается внешний модем. Порт COM2 можно использовать также и для связи двух компьютеров, если необходимо организовать двухмашинный комплекс. Для этого достаточно изготовить трехпроводный кабель, связывающий контакты 9- или 25-контактных разъемов так, как это показано на рис. 31.3, и соединить с его помощью свободные разъемы последовательных интерфейсов компьютеров.

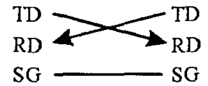


Рис. 31.3. Кабель для связи двух компьютеров через последовательные порты

Рассмотрим основы программирования последовательного интерфейса на примере двухмашинного комплекса. Программирование последовательной передачи данных в общем случае включает в себя два этапа: инициализацию интерфейса и передачу-прием данных. На этапе инициализации необходимо задать скорость передачи, а также характеристики кадра (число стоп-битов, наличие или отсутствие проверки четности и пр.). Разумеется, все эти настройки должны совпадать на передающем и принимающем оборудовании. Если связь будет работать в режиме прерываний, следует также определить условия возбуждения сигнала прерывания (например, по приему очередного символа или по его посылке). Настроив интерфейсы обеих машин, можно приступить к передаче данных. Здесь необходимо продумать вопросы взаимодействия передающего и принимающего оборудования, конкретно – как принимающая машина узнает о начале передачи и как она определит, что передача завершилась. В простейшем случае, когда число передаваемых байтов известно заранее, достаточно запустить принимающую программу раньше, чем передающую, и в обеих этих программах предусмотреть циклы передачи и приема из одинакового числа шагов.

Управление последовательным интерфейсом можно выполнить либо с помощью соответствующих функций прерывания 14h BIOS, либо на физическом уровне, обращаясь непосредственно к регистрам последовательных портов. Первый способ немного проще, но имеет ограниченные возможности, в частности не позволяет настроить режим прерываний. Управление последовательным портом через его регистры обеспечивает большую скорость и полноту. В приводимых ниже примерах использованы функции BIOS.

Для инициализации последовательного порта предусмотрена функция 00h прерывания 14h. Номер порта (0 – COM1, 1 – COM2) указывается в регистре DX; режим работы порта задается в регистре AL с помощью байта инициализации, назначение отдельных битов которого приведено на рис. 31.4.

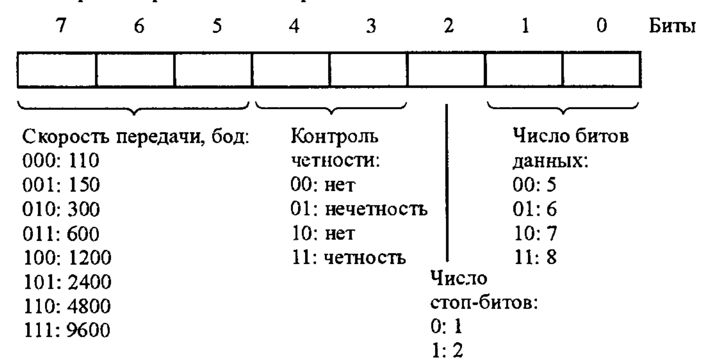


Рис. 31.4. Параметры инициализации последовательного порта

Передача 1 байта данных в линию осуществляется с помощью функции 01h, а прием байта данных – с помощью функции 02h. В обоих случаях в регистре DX указывается номер порта, а регистр AL служит источником и приемником байта данных. Если при приеме данных функцией 02h во входном регистре порта байт данных отсутствует, функция возвращает ненулевое значение в регистре AH; если байт принят, значение AH равно нулю.

Пример 31.1. Передача данных через последовательный порт

```
;Инициализируем порт передающего компьютера
        mov     AH,0h       ;Функция инициализации
        mov     AL,11101111b ;9600 бод, 2 стоп-бита, 8 байт
        mov     DX,1        ;COM2
        int     14h         ;Вызов BIOS
;Пошлем в цикле 10 байт
        mov     CX,10       ;Число передаваемых байтов
        mov     AL,40h      ;Пересылаемый байт
snd:    mov     AH,01h      ;Функция передачи
        mov     DX,1        ;COM2
        int     14h         ;Вызов BIOS
        loop    snd         ;Цикл
```

В приведенном примере для простоты 10 раз пересылается один и тот же код 40h (код символа @). Нетрудно модифицировать программу так, чтобы в каждом шаге цикла пересылаемый байт тем или иным способом изменял свое значение.

Пример 31.2. Прием данных через последовательный порт

```
;Инициализируем порт принимающего компьютера
        mov   AH,0h      ;Функция инициализации
        mov   AL,1110111b ;9600 бод, 2 стоп-бита, 8 байт
        mov   DX,1       ;COM2
        int   14h        ;Вызов BIOS
;Примем в цикле 10 байт
        mov   CX,10      ;Число принимаемых байтов
rcv:    mov   AH,02h     ;Функция приема
        mov   DX,1       ;COM2
        int   14h        ;Вызов BIOS
        cmp   AH,0       ;Байт пришел?
        jne   rcv        ;Нет, повторим попытку
        mov   AH,02h     ;Функция вывода на экран
        mov   DL,AL      ;Принятый байт
        int   21h        ;Вызов DOS
        loop  rcv        ;Цикл
```

В примере 31.1 предусмотрен вывод каждого принятого байта на экран для контроля. При этом предполагается, что по линии связи пересылаются отображаемые на экране коды.

Раздел третий

ОРГАНИЗАЦИЯ ПРОГРАММ

Статья 32. **Программы .EXE и .COM**

До сих пор, рассматривая различные программные примеры, мы помещали команды программы в сегмент команд, данные – в сегмент данных, а для операций со стеком предусматривали в программе отдельный сегмент стека. У читателя могло при этом создаться представление, что такая многосегментная структура программы является естественной или даже единственно возможной. Однако это не так.

Операционная система MS-DOS предусматривает два типа выполнимых программ, которым соответствуют расширения имен выполнимых файлов .COM и .EXE. Основное различие этих программ заключается в том, что программы типа .COM состоят из единственного сегмента, в котором размещаются программные коды, данные и стек, а в программах типа .EXE для собственно программы, данных и стека предусматривают отдельные сегменты. Таким образом, размер программы типа .COM не может превысить 64 Кбайт, а размер программы типа .EXE не имеет такого жесткого ограничения, так как в нее может входить любое число сегментов команд и данных. Все приводимые ранее примеры относились к программам типа .EXE.

Программы типа .EXE и .COM различаются форматом исходного текста, процедурой подготовки выполнимого файла, а также форматами загрузочных файлов.

Структура исходного текста типичной .EXE-программы выглядит следующим образом:

```
text    segment                 ;(1)Начало сегмента команд
assume CS:text,DS:data          ;(2)
myproc  proc                    ;(3)Начало главной процедуры
        mov     AX,text         ;(4)Инициализация сегментного
        mov     DS,AX           ;(5)регистра DS
        ...                     ;(6)Программные предложения
        mov     AX,4C00h        ;(7)Функция DOS 4Ch
        int     21h             ;(8)завершения программы
myproc  endp                    ;(9)Конец главной процедуры
text    ends                    ;(10)Конец сегмента команд
data    segment                 ;(11)Начало сегмента данных
        ...                     ;(12)Описания данных
data    ends                    ;(13)Конец сегмента данных
stk     segment stack           ;(14)Начало сегмента стека
        ...                     ;(15)Стек
stk     ends                    ;(16)Конец сегмента стека
        end     myproc          ;(17)Конец текста с точкой входа
```

В программе описаны три сегмента: команд text, данных data и стека stk. Собственно программа обычно состоит из процедур. Деление программы на процедуры не обязательно, но повышает ее структурированность и наглядность. В приведенном тексте сегмент команд содержит единственную процедуру myproc, открываемую оператором proc и закрываемую оператором endp. Перед обоими операторами указывается имя процедуры. Поскольку после трансляции имя процедуры приобретает значение

смещения первого предложения процедуры в сегменте команд, это имя можно указать в качестве точки входа в последнем предложении программы (оператор end).

Как уже отмечалось ранее, в процессе загрузки выполнимого модуля программы в память система пристраивает к началу программы дополнительный сегмент – префикс программы PSP размером точно 256 байт. Таким образом, образ программы типа .EXE в памяти всегда имеет на один сегмент больше, чем включено в нее программистом (рис. 32.1).

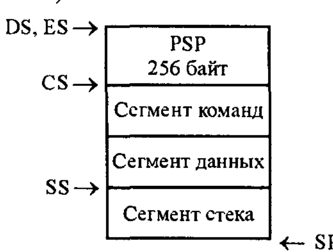


Рис. 32.1. Образ памяти программы типа .EXE

Система, загрузив программу в память, инициализирует сегментные регистры, так что регистры DS и ES указывают на начало PSP, CS – на начало сегмента команд, а SS – на начало сегмента стека. В указатель команд IP загружается смещение точки входа в программу (которое берется из операнда директивы end), а в указатель стека SP – смещение конца сегмента стека. Таким образом, после загрузки программы в память адресуемыми оказываются все сегменты, кроме сегмента (или сегментов) данных. Инициализация регистра DS в первых строках программы (предложения 4 и 5 предыдущего примера) позволяет сделать адресуемым и сегмент данных.

Формат .EXE наиболее удачно отвечает требованиям разработчиков программ и является преобладающим. Однако в некоторых случаях, когда объем программы невелик, оказывается удобно не дробить программу на отдельные сегменты, а включить все компоненты программы в один сегмент. Этот единственный сегмент должен, таким образом, содержать префикс программы (PSP), коды команд, данные и стек. Такие односегментные программы, соответствующие в терминологии языков высокого уровня минимальной (tiny) модели памяти, обычно образуют загрузочные модули типа .COM, хотя иногда могут иметь и другие расширения (например, SYS). Программы типа .COM не имеют особых преимуществ перед программами типа .EXE, кроме своей компактности, однако они широко используются, прежде всего в качестве резидентных программ.

При создании программы типа .COM необходимо выполнение двух условий: во-первых, исходный текст программы должен быть написан в определенном формате, с ограничениями, соответствующими минимальной модели памяти, и, во-вторых, необходимо после компоновки получить выполнимый файл с расширением .COM. Последнее выполняется по-разному в различных системах программирования.

При использовании пакета TASM достаточно при вызове компоновщика указать ключ /T:

```
TASM /Z /N P, P, P
TLINK /X /T P, P
```

Обратите внимание на отсутствие ключей /ZI и /V в строках вызова ассемблера и компоновщика соответственно. Эти ключи, передающие в выполнимый файл отладочную информацию, неприменимы в случае программ типа .COM, отладка которых не-

сколько затрудняется, так как программист, работая с отладчиком, видит в кадре отладчика только коды команд с их мнемоникой, но не исходный текст программы.

После выполнения приведенных выше команд в дополнение к исходному файлу P.ASM будут образованы еще три файла: объектный P.OBJ, с листингом трансляции P.LST и выполнимый P.COM.

При использовании ассемблера MASM и соответствующего ему компоновщика LINK процесс подготовки .COM-программы усложняется. Компоновщик LINK может создать загрузочный модуль лишь с расширением .EXE. Если, однако, исходная программа написана в формате .COM, этот модуль .EXE не является полноценной программой и в большинстве случаев будет неработоспособен. Чтобы получить выполнимую программу, файл с расширением .EXE необходимо преобразовать в формат .COM с помощью утилиты EXE2BIN, входящей с состав пакета MS-DOS:

```
EXE2BIN P.EXE P.COM
```

Для того чтобы освоить написание программ в формате .COM, воспользуемся самой первой программой из статьи 1 и видоизменим ее требуемым образом (пример 32.1).

Пример 32.1. Простейшая программа типа .COM

```
text     segment
assume   CS:text,DS:text
         org 256              ;Резервирование места для PSP
myproc   proc
         mov   AH,09h
         mov   DX,offset mesg
         int   21h
         mov   AX,4C00h
         int   21h
myproc   endp
mesg     db 'Программа типа .COM$'
text     ends
         end   myproc
```

Программа содержит единственный сегмент text. В директиве assume указано, что сегментные регистры CS и DS будут соответствовать этому сегменту (в этом конкретном примере, в котором нет прямых обращений к ячейкам данных, можно было ограничиться директивой assume CS:text).

Директива org 256 резервирует 256 байт для PSP. Заполнять PSP будет по-прежнему система, но место под него в начале сегмента должен отвести программист. В программе нет необходимости инициализировать регистр DS, поскольку его, как и остальные сегментные регистры, инициализирует система. Данные можно разместить после программной процедуры, как это сделано в нашем примере, или внутри нее, или даже перед ней. Следует только иметь в виду, что при загрузке программы типа .COM регистр IP всегда инициализируется числом 256 (100h), поэтому сразу за директивой org 256 должно стоять первое выполнимое предложение программы. Если в начале программы желательно расположить данные, перед ними следует поместить команду перехода на первую команду программы:

```
myproc   proc
         jmp   entry
mesg     db 'Программа типа .COM$'
entry:   mov   AH,09h
         mov   DX,offset mesg
         int   21h
         ...
```

Образ памяти программы типа .COM показан на рис. 32.2.

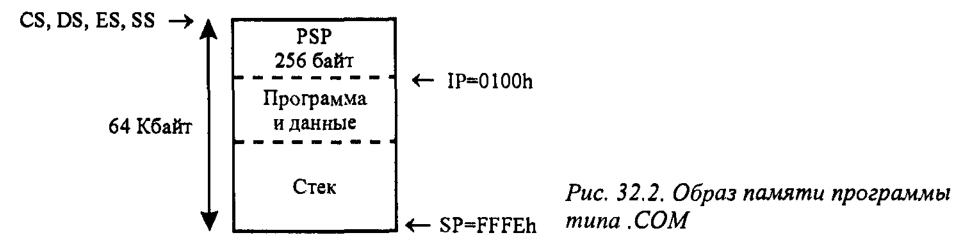


Рис. 32.2. Образ памяти программы типа .COM

После загрузки программы все 4 сегментных регистра указывают на начало единственного сегмента, т. е. фактически на начало PSP. Указатель стека автоматически инициализируется числом FFFEh. Таким образом, независимо от фактического размера программы, ей выделяется 64 Кбайт адресного пространства, всю нижнюю часть которого занимает стек.

Создайте файл с программой из примера 32.1. Подготовьте программу к выполнению. Выполните пробный прогон программы и убедитесь, что она работает как ожидалось. Запустите программу под управлением отладчика. Внимательно рассмотрите содержимое регистров процессора: сегментных, указателя команд, указателя стека. Вы увидите, что все обстоит в точности так, как показано на рис. 32.2.

Что же в этом хорошего? Мы написали очень короткую программу, содержательная часть которой занимает (как можно увидеть из листинга трансляции) всего 20h байт, а с учетом PSP – 120h = 288 байт. Между тем в памяти эта программа займет 64 Кбайт, причем большая часть этого пространства отводится под стек. Однако в реальной программе, если только мы не собираемся хранить в стеке большие массивы данных, размер стека можно ограничить величиной 200...300 байт. Выполним эту операцию (пример 32.2).

Пример 32.2. Программа типа .COM с явным выделением стека

```
text    segment             ;(1)
assume  CS:text,DS:text     ;(2)
        org 256             ;(3)Резервирование места для PSP
myproc  proc                ;(4)
        mov   SP,offset endstk ;(5)Инициализация SP
        mov   AH,09h        ;(6)
        mov   DX,offset mesg ;(7)
        int   21h           ;(8)
        mov   AX,4C00h      ;(9)
        int   21h           ;(10)
myproc  endp                ;(11)
mesg    db 'Программа типа .COM$' ;(12)
        evendata            ;(13)Сместимся к границе слова
        org $+256           ;(14)Резервируем место для стека
endstk=$                    ;(15)Обозначаем смещение этой точки
text    ends                ;(16)
        end   myproc        ;(17)
```

В самом конце программного сегмента с помощью директивы org (предложение 14) сместим счетчик текущего адреса на 256 байт относительно текущего состояния. В этом пустом пространстве будет располагаться стек. Директива evendata выравнивает конец области стека на четный (even) адрес, т. е. на границу слова, что делает работу стека чуть более наглядной. Необходимости в этом нет; заполнение стека может начинаться и с нечетного адреса. В предложении (15) вводится символическое

обозначение endstk, которому дается значение смещения первого байта за пространством стека (для приведенного выше примера это значение составляет 224h). Именно этим смещением следует инициализировать указатель стека, что и выполняется первой командой процедуры myproc (предложение 5). Теперь программа в процессе своей работы не выйдет за пределы адресного пространства 224h = 548 байт, и мы действительно получили компактную односегментную программу.

В действительности, однако, выполненная выше операция почти бессмысленна. Дело в том, что MS-DOS принципиально является однозадачной операционной системой, которая позволяет в каждый момент времени выполняться лишь одной программе. Запустить одновременно две или больше программ нельзя – в DOS нет таких средств. В этом случае вопрос о размере программы теряет свою остроту. Не имеет значения, будет ли наша программа занимать 548 байт или 64 Кбайт; в любом случае вся оперативная память за пределами программы никак использоваться не будет.

Между прочим, отражением однозадачности DOS является тот факт, что при загрузке программы в память DOS, независимо от фактического размера программы, выделяет для нее всю свободную память (т. е. около 600 Кбайт). Поставим эксперимент по проверке этого утверждения. Попробуем с помощью соответствующей системной функции выделить для нашей программы дополнительную память. Эта операция выполняется с помощью функции 48h, при вызове которой регистр BX должен содержать число требуемых параграфов памяти. Если функция выполнилась успешно, в регистре AX возвращается сегментный адрес выделенного участка (блока) памяти. Если выделить затребованный объем памяти не удалось, система фиксирует ошибку: устанавливается флаг CF, в регистре AX возвращается код ошибки (в данном случае код 8 – нехватка памяти), а в регистре BX – размер наибольшего свободного блока памяти в параграфах.

Включим в любое место программы 32.2 (например, после предложений 5 или 8) следующие строки:

```
mov   AH,48h        ;Функция выделения блока памяти
mov   BX,1          ;Затребуем 1 параграф (16 байт)
int   21h           ;Переход в MS-DOS
```

Запустим программу в отладчике, остановив ее после выполнения последней команды приведенного выше фрагмента. В этой точке мы получим CF=1, AX=8, BX=0, что свидетельствует о полном отсутствии в системе свободной памяти и невозможности выделения даже одного параграфа.

Какой же смысл может иметь сокращение размера .COM-программы? Выше уже упоминалось, что программы такого типа часто используются в качестве резидентных. Резидентная программа после ее запуска остается в памяти, но возвращает управление DOS, давая возможность оператору запустить обычную (нерезидентную) программу. Далее выполняется эта запущенная программа, роль же резидентной обычно заключается в обработке тех или иных аппаратных прерываний (от клавиатуры, таймера и т. д.), что не требует много времени и не очень тормозит выполнение основной программы. Ясно, что в этом случае резидентную программу следует составлять таким образом, чтобы она занимала в памяти лишь столько место, сколько ей нужно для нормальной работы. Вопросы разработки и использования резидентных программ будут освещены в последующих статьях этого раздела.

Как было показано выше, выполнимый модуль любой программы начинается с префикса программы PSP, специальной системной области размером 100h=256 байт, в которой система сохраняет некоторые данные, имеющие отношение к выполняемой программе. В определенных ситуациях (например, при работе с файлами) система обращается к PSP, используя хранящиеся там данные. Ряд полей PSP в современных версиях MS-DOS потеряли свое значение, другие же, наоборот, очень важны, и их назначение желательно понимать, тем более что в ряде случае они используются не только системой, но и прикладными программистами. В табл. 32.1 приведено описание наиболее важных полей PSP. Более подробно их назначение будет поясняться в последующих главах по мере описания соответствующих средств DOS.

Таблица 32.1. Содержимое префикса программы PSP

Смещение от начала PSP	*Размер, байт*	*Описание*
02h	2	Сегментный адрес первого байта за памятью, отведенной программе
0Ah	4	Адрес перехода в COMMAND после завершения программы (вектор 22h)
0Eh	4	Адрес обработчика Ctrl+C (вектор 23h)
12h	4	Адрес обработчика критической ошибки (вектор 24h)
16h	2	Сегмент родительского PSP
18h	20	Таблица файлов задания JFC
2Ch	2	Сегмент блока окружения программы
2Eh	4	SS:SP при входе в последний вызов int 21h
32h	2	Размер JFC (по умолчанию 20)
34h	4	Адрес JFC программы (по умолчанию PSP:18h)
40h	2	Версия DOS (например, 0614h=6.22)
80h	128	Параметры командной строки при запуске программы

Рассмотрим вкратце отличия загрузочных модулей программ .EXE и .COM, а также процесс их загрузки в память.

Если взглянуть на листинг трансляции рассмотренного ранее примера 32.1 (рис. 32.3), то можно сделать вывод, что программа имеет размер (включая место для PSP) 120h=288 байт. Действительно, именно столько места займет содержательная часть программы в памяти, когда она будет загружена для выполнения. При этом образ программы в памяти будет в точности соответствовать тексту листинга за тем исключением, что в листинге только выделено место для PSP, а при загрузке программы в память система заполнит это место соответствующим содержимым. Программы типа .COM не требуют никакой настройки, отчего их загрузка осуществляется относительно быстро. Разумеется, система, загрузив программу в память и определив таким образом ее местоположение, должна настроить ряд регистров процессора, конкретно – поместить во все 4 сегментных регистра сегментный адрес начала программы, в регистр IP – число 100h, а в регистр SP – FFFEh; однако регистры находятся в процессоре и модифицируются почти мгновенно, а содержимое сегмента программы при загрузке не изменяется.

```
                                 ;Пример 32.1. Программа типа .COM
  0000                           text    segment
                                 assume  CS:text,DS:text
                                         org     256
  0100                           main    proc
  0100   B4 09                           mov     AH,09h
  0102   BA 010Cr                        mov     DX,offset mesg
  0105   CD 21                           int     21h
  0107   B8 4C00                         mov     AX,4C00h
  010A   CD 21                           int     21h
  010C                           main    endp
  010C   8F E0 AE A3 E0 A0 AC+   mesg    db      'Программа типа .COM$'
         AC A0 20 E2 A8 AF A0+
         20 2E 43 4F 4D 24
  0120                           text    ends
                                         end     main
```

Рис. 32.3. Листинг примера 32.1

Между прочим, если обратиться к выполнимому файлу примера 32.1 (пусть он получил имя 32-01.COM), то окажется, что его длина составляет всего 20h=32 байт, т. е. в выполнимый файл не включается место для PSP (рис. 32.4), что, впрочем, естественно. Конкретное содержимое PSP система определяет в процессе загрузки программы в память, и в выполнимом файле этого содержимого быть не может. Выполнимый файл включает в себя лишь программные строки вместе со строками данных.

```
00000  B4 09 BA 0C | 01 CD 21 B8 | 00 4C CD 21 | 8F E0 AE A3   ┤.║+☺=!╕.L=!Прог
00010  E0 A0 AC AC | A0 20 E2 A8 | AF A0 20 2E | 43 4F 4D 24   рамма типа .COM$
```

Рис. 32.4. Содержимое файла 32-01.COM

Не так обстоит дело для программ типа .EXE. Выполнимый файл типа .EXE содержит, кроме кодов сегментов команд, данных и, возможно, стека, еще и заголовок, состоящий из одного или нескольких блоков размером 512 байт каждый. В этом заголовке, создаваемом в процессе трансляции и компоновки программы, хранится информация, необходимая системе для правильной настройки регистров процессора и самой программы при ее загрузке в память. Формат заголовка приведен в табл. 32.2.

Таблица 32.2. Формат заголовка выполнимого модуля .EXE

Смещение	*Число байтов*	*Описание*
00h	2	Признак файла типа .EXE (коды символов MZ)
02h	2	Размер файла .EXE с программой по модулю 512
04h	2	Размер файла .EXE с программой в блоках по 512 байт
06h	2	Число элементов таблицы настройки
08h	2	Размер заголовка в параграфах
0Ah	2	Минимальное число параграфов, требуемых программе дополнительно к ее образу на диске (при наличии в программе неинициализированных сегментов)
0Ch	2	Максимальное число параграфов, требуемое программе дополнительно к ее образу на диске (по умолчанию FFFFh)
0Eh	2	Смещение сегмента стека от начала программы в параграфах
10h	2	Содержимое SP при входе в программу
12h	2	Контрольная сумма

14h	2	Содержимое IP при входе в программу
16h	2	Смещение сегмента команд от начала программы в параграфах
18h	2	Смещение первого элемента настройки
1Ah	2	Число перекрытий
1Ch	...	Таблица настройки переменной длины

Как можно увидеть из приведенной таблицы, в заголовке файла .EXE содержится информация, которую система использует при загрузке программы в память: значение IP, которое поступает в заголовок из параметра заключительной директивы end, значения CS и SS, которые можно вычислить прибавлением содержимого ячеек заголовка 16h и 10h к начальному сегментному адресу программы, и т. д. В таблице настройки содержатся сведения об изменении тех адресов в программе, для которых значения в листинге и в загруженной программе различаются.

Между прочим, в заголовке отсутствует информация о настройке сегментных регистров данных DS и ES. Действительно, система не может знать, как программист пожелает настроить эти сегментные регистры по ходу выполнения программы, хотя бы потому, что число сегментов данных в программе может превышать число сегментных регистров процессора. Как уже отмечалось выше, при загрузке программы в память регистры DS и ES указывают на начало префикса программы PSP. Занесение в них сегментных адресов сегментов или областей адресуемых данных является обязанностью программиста.

Статья 33. **Директива assume**

Обсудим подробнее роль директивы ассемблера assume. Типичная программа типа .EXE имеет в качестве одной из первых строк предложение

```
assume   CS:text,DS:data
```

в котором устанавливается соответствие сегментного регистра CS сегменту команд text, а сегментного регистра DS – сегменту данных data.

Директивы assume, которых в программе может быть любое количество, в принципе могут располагаться в любом месте программы, однако обязательно до тех программных строк, в которых выполняется обращение к описываемым в конкретной директиве сегментам. Поэтому предложение вида

```
assume CS:text
```

должно стоять в программе до любых программных строк и даже до объявления процедур. Что же касается установки соответствия регистра DS и конкретного сегмента данных, то, если это соответствие не указано в первой директиве assume, оно может быть описано и позже отдельной директивой:

```
assume DS:data
```

Каково же назначение директивы assume? Что значит "соответствие сегментного регистра сегменту"? Выше уже отмечалась исключительно важная роль сегментных регистров при обращении к ячейкам памяти. Физический 20-разрядный адрес любой ячейки памяти вычисляется процессором путем сложения умноженного на 16 сегментного адреса, хранящегося в одном из сегментных регистров, со смещением, указанным в коде команды. Таким образом, при каждом обращении к памяти обязательно исполь-

зуется тот или иной сегментный регистр; обратиться к памяти без использования сегментного регистра невозможно.

Для того чтобы процессор при выполнении команды знал, каким сегментным регистром ему пользоваться, код сегментного регистра включается в полный код команды в виде префикса. Каждый сегментный регистр имеет свой код префикса, хотя во многих случаях отсутствие префикса обозначает необходимость использования конкретно регистра DS. Так, команда

```
        mov     AX,mem
```

преобразуется в код A1 *dddd*, если адресация должна осуществляться через сегментный регистр DS, и в код 26 A1 *dddd*, если сегментный адрес следует извлекать из регистра ES. Здесь 26 – префикс команды, обозначающий регистр ES, а *dddd* – конкретная величина смещения ячейки mem от начала того сегмента, в котором она расположена.

Директива assume как раз и служит для того, чтобы указать транслятору, как кодировать команды прямого обращения к памяти. Пусть в программу входит сегмент данных с единственной ячейкой mem. Смещение этой ячейки будет равно 0000h:

```
data    segment
mem     dw 1234h
data    ends
```

Пусть, далее, в сегменте команд имеется команда

```
        mov     AX,mem
```

Если к моменту трансляции этой команды действует директива

```
assume DS:data
```

то ассемблер преобразует эту команду в код A1 0000. Процессор, не увидев перед кодом команды префикса определения сегмента, будет при ее выполнении использовать регистр DS. Если же была выполнена директива

```
assume ES:data
```

то в объектный модуль будет включен код 26 A1 0000 и процессор при выполнении этой команды будет извлекать сегментный адрес из регистра ES.

Можно ли во втором случае обращаться к ячейкам сегмента данных с использованием сегментного регистра DS? Можно, но для этого во все команды прямого обращения к памяти следует включить префикс явной замены сегмента:

```
        mov     AX,DS:mem
```

Такая команда, независимо от формы директивы assume, будет транслироваться в код, предполагающий использование регистра DS. Точно так же, если действует директива assume DS:data, а мы хотим адресоваться к ячейке сегмента data через регистр ES, необходимо использовать префикс ES:

```
        mov     AX,ES:mem
```

Таким образом, директива assume определяет, префикс какого сегментного регистра будет включаться в код команды по умолчанию, хотя во всех случаях это умолчание можно отменить, включив в команду префикс явной замены сегмента.

Все сказанное выше относится только к правилам трансляции предложений программы и никак не определяет истинного содержимого того или иного сегментного регистра к моменту выполнения команды обращения к памяти. Во всех случаях перед обращением к памяти необходимо загрузить в сегментный регистр сегментный адрес

адресуемого сегмента. Как мы видели, инициализация сегментного регистра осуществляется обычно с помощью пары команд

```
        mov     AX,data
        mov     DS,AX
```

или

```
        mov     AX,data
        mov     ES,AX
```

если к сегменту data предполагается обращаться через регистр ES.

Директива assume может использоваться внутри программы неоднократно. Предположим, что на некотором достаточно продолжительном участке программы к определенным полям данных надо обращаться через сегментный регистр DS, а на другом участке – через регистр ES. Тогда между этими участками можно сменить соответствие сегментных регистров сегментам данных, как это схематически показано ниже:

```
;Сегмент данных
data    segment
mem1    dw      ?
mem2    dw      ?
data    ends
;Программные строки в сегменте команд
        assume DS:data
        mov    AX,mem1      ;Команда выполняется как mov AX,DS:mem1
        mov    BX,mem2      ;Команда выполняется как mov BX,DS:mem2
        assume DS:nothing
        assume ES:data
        mov    AX,mem1      ;Команда выполняется как mov AX,ES:mem1
        mov    BX,mem2      ;Команда выполняется как mov BX,ES:mem2
```

Директива assume DS:nothing (nothing, ничего) снимает закрепление за сегментом data регистра DS и позволяет закрепить за ним другой регистр, ES. Того же эффекта можно было достигнуть, закрепив регистр DS за каким-то другим сегментом (если, конечно, он имеется). Необходимо только перед описанием соответствия регистра ES и сегмента data отменить закрепление за этим сегментом регистра DS.

Очевидно, что, независимо от директивы assume, перед первым участком необходимо настроить на сегмент data регистр DS, а перед вторым – ES.

Статья 34. **Подпрограммы**

Подпрограммы являются важнейшим средством любого языка программирования. Лишь самые простые программы имеют линейную структуру, в которой отсутствуют повторения. Как правило, программа по ходу своего выполнения неоднократно обращается к определенным алгоритмам, например преобразования чисел или строк, анализа данных, ввода с клавиатуры или вывода на экран и т. д. Такие алгоритмы разумно оформлять в виде подпрограмм и вызывать из основной программы по мере необходимости, что может существенно уменьшить объем программы и повысить ее наглядность.

Остановимся на особенностях организации и выполнения подпрограмм. Начнем с рассмотрения ближних подпрограмм, которые располагаются в одном программном сегменте с вызывающей процедурой. Организация многосегментных программных комплексов будет рассмотрена в следующей статье.

Рассмотрим для примера простой программный комплекс, в котором многократно в цикле вызывается подпрограмма, осуществляющая программную задержку, после чего на экран выводится короткая строка символов. Величина задержки (в условных единицах) передается в подпрограмму в регистре AX в качестве параметра.

Алгоритм программной задержки уже рассматривался в статье 8; в примере 34.1 этот алгоритм будет оформлен в виде процедуры-подпрограммы delay.

Пример 34.1. Программная задержка с помощью подпрограммы

```
text    segment
        assume CS:text,DS:data
;Процедура-подпрограмма. При входе AX=величина задержки
delay   proc
        push   CX           ;Сохраним используемый регистр
        mov    CX,AX        ;Перенесем параметр в CX
outer:  push   CX           ;Сохраним счетчик внешнего цикла
        mov    CX,65535     ;Число шагов внутреннего цикла
inner:  loop   inner        ;Внутренний цикл - 1 команда
        pop    CX           ;Восстановим внешний счетчик
        loop   outer        ;Повторение параметр раз
        pop    CX           ;Восстановим сохраненный регистр
        ret                 ;Возврат из подпрограммы
delay   endp
;Главная процедура, с которой начинается выполнение программы
main    proc
        mov    AX,data
        mov    DS,AX
        mov    CX,10        ;Цикл из 10 шагов
cycle:  mov    AX,200       ;Параметр, передаваемый в подпрограмму
        call   delay        ;Команда вызова подпрограммы
        mov    AH,09h       ;Вывод на экран трех символов
        mov    DX,offset string
        int    21h
        loop   cycle        ;Цикл из CX шагов
        mov    AX,4C00h     ;Завершение программы
        int    21h
main    endp
text    ends
data    segment
string  db '<> $'
data    ends
stk     segment stack
        db     256 dup (0)
stk     ends
        end    main
```

Программа состоит из двух процедур – главной с именем main и процедуры-подпрограммы delay. Каждая процедура начинается оператором proc, перед которым указывается имя процедуры, а заканчивается оператором endp (end procedure, конец процедуры). Порядок процедур в тексте программы не имеет значения, однако имя главной процедуры, с которой начинается выполнение программы, должно быть указано в качестве операнда директивы end, завершающей текст программы.

Подпрограммы вызываются командой call (вызов); каждая подпрограмма должна заканчиваться командой ret (return, возврат), которая передает управление в точку возврата, т. е. на команду вызывающей программы, следующую за командой call.

Деление программы на процедуры не является обязательным. Выше уже отмечалось, что директивы ассемблера, такие, как proc или endp, не находят своего отраже-

ния в загрузочном модуле и никак не влияют на ход выполнения программы. Фактически они нужны лишь для повышения наглядности исходного текста. С таким же успехом можно было ликвидировать деление программы на процедуры, а в качестве точек входа использовать метки:

```
delay:
        ...                     ;Текст подпрограммы
        ret
main:   mov    AX,data
        mov    DS,AX
        ...
        call   delay
        ...
        mov    AX,4C00h         ;Завершение программы
        int    21h
        ...
        end    main
```

Главная программа и подпрограммы могут располагаться в исходном тексте в любом порядке, причем процедуры-подпрограммы можно вкладывать друг в друга. Например, возможен такой вариант компоновки примера 34.1:

```
text    segment                 ;Начало сегмента команд
main    proc                    ;Объявление главной процедуры
        ...                     ;Текст главной процедуры
        mov    AX,4C00h         ;Завершение программы
        int    21h
delay   proc                    ;Объявление процедуры-подпрограммы
        ...                     ;Текст подпрограммы
        ret                     ;Возврат из подпрограммы
delay   endp                    ;Конец процедуры-подпрограммы
main    endp                    ;Конец главной процедуры
text    ends                    ;Конец сегмента команд
        ...
        end    main             ;Конец текста и точка входа
```

Здесь процедура-подпрограмма delay располагается внутри главной процедуры main после строк завершения программы (вызов функции DOS 4Ch). Последний момент является принципиально важным. Подпрограммы следует располагать таким образом, чтобы они не могли начать выполняться "сами по себе", без вызова командой call. Приведенный пример удовлетворяет этому требованию. Действительно, вызовом функции завершения 4Ch управление передается системе, в нашу программу уже никогда не вернется, и строки, стоящие после команды int 21h, выполняться не будут. В то же время приведенный ниже пример грубо неверен:

```
text    segment                 ;Начало сегмента команд
main    proc                    ;Объявление главной процедуры
delay   proc                    ;Объявление процедуры-подпрограммы
        ...                     ;Текст подпрограммы
        ret                     ;Возврат из подпрограммы
delay   endp                    ;Конец процедуры-подпрограммы
        ...                     ;Текст главной процедуры
        mov    AX,4C00h         ;Завершение программы
        int    21h
main    endp                    ;Конец главной процедуры
text    ends                    ;Конец сегмента команд
        ...
        end    main             ;Конец текста и точка входа
```

В такой программе после ее активизации сразу же начнется выполнение процедуры delay. Страшно здесь не то, что эта процедура выполнится "вне очереди" (просто в начале программы произойдет небольшая задержка), а то, что она завершится выполнением команды ret, обратной по отношению к команде call. Однако команды call у нас не было, и такая "непарная" команда ret приведет к "зависанию" системы. Для того, чтобы понять, что здесь происходит, надо разобраться в механизмах вызова подпрограмм и возврата из них.

На рис. 34.1 приведен фрагмент листинга трансляции программы 34.1 с указанием кодов команд, их смещений, мнемонических обозначений и описанием их действия.

```
Смеще-  Код       Предложения          Действие
ние     команды   программы            команды

0000              delay proc
0000    51              push CX
                        ...
000D    C3              ret  ---------> Из стека адрес возврата (001Ch) в IP
000E              delay endp

000E              main  proc
000E    B8 1138         mov  AX,data
0011    8E D8           mov  DS,AX
                        ...
0019    E8 FFE4         call delay  --> 1. Содержимое IP (001Ch) в стек
001C    B4 09           mov  AH,09h     2. Смещение вызываемой процедуры,
                        ...                равное 001Ch + FFE4h = 0000h, в IP
002A              main  endp
```

Рис. 34.1. Фрагмент листинга трансляции примера 34.1 с поясняющей информацией

Сегмент команд начинается у нас с процедуры delay. Первая команда этой процедуры push CX имеет поэтому смещение 0000h. Процедура delay занимает Eh=14 байт с относительными адресами команд от 0000h до 000Dh. Последней командой процедуры delay является 1-байтовая команда ret с кодом C3h.

За процедурой delay располагается главная процедура main. Ее первая команда mov AX,data имеет смещение 000Eh. Код команды включает код операции mov (B8h) и значение имени data, равное сегментному адресу сегмента данных. При загрузке программы под управлением отладчика TD сегментный адрес data оказался равным 1138h.

Команда call delay расположена по адресу 0019h. В ее полный код входит код операции call (E8h) и адрес процедуры delay, на которую надо осуществить переход. Этот адрес записан в виде смещения к началу процедуры delay относительно текущего содержимого IP, т. е. относительно адреса следующей команды (в нашем случае команды mov AH,09h). Смещение это знаковое и в данном случае отрицательное, так как процедура delay располагается до процедуры main. Поскольку адрес delay равен нулю, а адрес следующей команды равен 1Ch, в коде команды записано число –1Ch, которое по правилам записи отрицательных чисел выражается кодом FFE4h.

Главная процедура занимает 1Ch=28 байт, а первый свободный байт после конца этой процедуры имеет смещение 002Ah. На этом заканчивается сегмент команд. С ближайшего адреса, кратного 16 (11380h в нашем случае), начинается сегмент данных, за которым следует сегмент стека.

Вернемся к рассмотрению команд call и ret. При выполнении команды call процессор помещает в стек адрес следующей команды, т. е. адрес возврата, а в IP заносит от-

носительный адрес вызываемой процедуры, который находится суммированием текущего содержимого IP и смещения, записанного в коде команды call. В результате указатель стека SP смещается вверх на одно слово (рис. 34.2), а процессор переходит на выполнение подпрограммы.

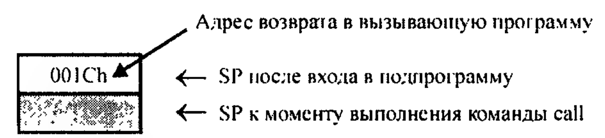


Рис. 34.2. Состояние стека после входа в подпрограмму

Команда ret выполняет обратную операцию – извлекает из верхнего слова стека (с восстановлением исходного состояния SP) адрес возврата и загружает его в IP, в результате чего процессор возвращается к выполнению вызывающей процедуры.

Из сказанного ясно, что, если в подпрограмме используется стек, с ним надо работать очень аккуратно: все, что заносится в стек в процессе выполнения подпрограммы, должно быть обязательно снято с него до выполнения команды ret, иначе эта команда извлечет из стека и загрузит в IP не адрес возврата, а какое-то данное, что заведомо приведет к нарушению выполнения программы.

Рассмотренный нами вызов подпрограммы носит название прямого ближнего или внутрисегментного вызова. Прямым такой вызов называется потому, что адрес перехода хранится непосредственно в коде команды (а это, в свою очередь, получилось потому, что мы указали в качестве операнда команды call имя подпрограммы). Если бы адрес подпрограммы хранился в каком-то другом месте (а именно в регистре или в ячейке памяти), то вызов был бы косвенным. Мы столкнемся с косвенными вызовами подпрограмм в последующих статьях. Вторая характеристика вызова говорит о том, что вызываемая подпрограмма находится в том же сегменте, что и вызывающая процедура. В этом случае для перехода на подпрограмму надо знать лишь половину полного адреса подпрограммы, а именно смещение точки перехода. Сегментный адрес остается тем же; он не фигурирует в строке вызова подпрограммы и отсутствует в коде команды. В следующей статье мы рассмотрим и другой вид подпрограмм – дальние подпрограммы, для обращения к которым следует применять межсегментные вызовы.

Статья 35. **Дальние подпрограммы**

Мы уже знаем, что программы типа .EXE могут содержать по несколько сегментов команд и данных. Это позволяет довести общий объем прикладной программы по крайней мере до размеров свободной оперативной памяти, т. е. до величины около 600 Кбайт.

При наличии в программе большого объема данных их придется раздробить на отдельные сегменты размером не более 64 Кбайт каждый и работать с ними поочередно, настраивая сегментные регистры DS и ES на начальные адреса текущей пары сегментов. Если же программа содержит большой объем вычислений, то самое разумное в одном сегменте расположить главную процедуру, а в остальных – процедуры-подпрограммы, которые можно будет вызывать как из главной процедуры, так и при необходимости друг из друга.

Чтобы продемонстрировать технику дальнего вызова подпрограмм, преобразуем приведенный в предыдущей статье пример так, чтобы программный комплекс состоял из двух сегментов команд (пример 35.1).

Пример 35.1. Дальний вызов подпрограммы

```
text1   segment             ;Первый сегмент команд
        assume  cs:text1,DS:data
main    proc                ;Главная процедура
        mov     AX,data
        mov     DS,AX
        mov     CX,10
cycle:  mov     AX,200
        call    far ptr delay
        mov     AH,09h
        mov     DX,offset string
        int     21h
        loop    cycle
        mov     AX,4C00h
        int     21h
main    endp
text1   ends                ;Конец первого сегмента команд
text2   segment             ;Второй сегмент команд
        assume  cs:text2    ;Оператор assume
delay   proc far            ;Дальняя процедура
        push    CX
        mov     CX,AX
outer:  push    CX
        mov     CX,65535
inner:  loop    inner
        pop     CX
        loop    outer
        pop     CX
        ret                 ;Дальний возврат
delay   endp
text2   ends                ;Конец второго сегмента команд
data    segment
string  db '<> $'
data    ends
stk     segment stack
        db      256 dup (0)
stk     ends
        end     main
```

В программе описаны два сегмента команд – text1 с процедурой main и text2 с процедурой-подпрограммой delay. Сегменты данных data и стека stk остались без изменений. Практически без изменений остались и тексты обеих процедур за исключением того, что процедура delay объявлена с описателем far (дальняя), а ее вызов в главной процедуре сопровождается описателем far ptr (far pointer, дальний указатель). Что изменилось при этом в программах процедур и чем оператор call far ptr отличается от простого call?

Рассмотрим, как и в предыдущей статье, фрагменты листинга трансляции программы с указанием кодов команд, их смещений, мнемонических обозначений и описанием их действия (рис. 35.1).

```
Смеще- Код            Предложения                   Действие
ние    команды        программы                     команды
0000                  text1 segment
0000                  main  proc
0000   B8 1138              mov   AX,data
                            ...
000B   9A 0000 1137         call far ptr delay  -->  1. CS=1135h в стек
0010   B4 09                mov   AH,09              2. IP=0010h в стек
                            ....                     3. 1137h из кода команды в CS
001E                        main endp                4. 0000h из кода команды в IP
001E                  text1 ends

0000                  text2 segment
0000                  delay proc far
0000   51                   push CX
                            ...
000D   CB                   ret  ------->  1. Из стека смещение возврата (0010h) в IP
000E                  delay endp           2. Из стека сегмент возврата (1135h) в CS
000E                  text2 ends
```

Рис. 35.1.Фрагменты листинга примера 35.1 с поясняющей информацией

Команда call far ptr delay расположена по относительному адресу 000Bh. В ее код входит код операции дальнего вызова call far ptr (9Ah) и полный адрес процедуры delay, на которую надо осуществить переход. Этот адрес записан в виде двух слов: смещения процедуры delay в том сегменте, где она расположена (0000h) и сегментного адреса этого сегмента, который, как выяснилось после загрузки программы в память, равен 1137h. Таким образом, процессор, считав из памяти код команды, имеет полную информацию о том, куда надо осуществить переход.

При выполнении команды call far ptr процессор помещает в стек два слова: сначала сегментный адрес текущего сегмента text1, который у нас оказался равен 1135h, затем – смещение возврата (содержимое IP, в данном случае 0010h). Следует обратить внимание на порядок записи в память компонентов двухсловного адреса: всегда в слово памяти с большим адресом записывается сегментный адрес, а в слово памяти с меньшим адресом – смещение. После сохранения в стеке адреса возврата процессор заносит в сегментный регистр команд CS сегментный адрес процедуры delay, а в IP – смещение delay. Оба эти значения процессор извлекает из кода команды call. В результате указатель стека смещается вверх на два слова (рис. 35.2), а процессор переходит на выполнение подпрограммы из другого сегмента.

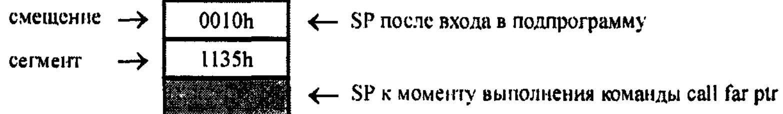


Рис. 35.2. Состояние стека после входа в дальнюю подпрограмму

Команда ret процедуры delay, расположенная по адресу 000Dh в сегменте text2, выполняет обратную операцию – снимает со стека два верхних слова, загружая первое в IP, а второе – в CS, в результате чего процессор возвращается к выполнению вызывающей процедуры.

Так осуществляется дальний, или межсегментный, вызов процедуры, расположенной в другом сегменте команд.

Почему же команда ret процедуры delay в примере 34.1 снимает со стека два слова, в то время как такая же на первый взгляд команда той же процедуры в предыдущем примере снимает только одно? В действительности, однако, эти две команды не эквивалентны. Сравнив строки с командами ret этих двух программ, можно заметить, что машинные коды команд ret различаются. Так получилось потому, что процедура delay во втором примере объявлена нами с помощью описателя far дальней. Транслятор воспринимает это объявление как требование преобразовать мнемоническое обозначение ret, встретившееся в этой процедуре, в код команды дальнего возврата CBh. Процедура delay первого примера не имеет явного описателя типа и по умолчанию рассматривается транслятором как ближняя. С целью повышения наглядности программы ее можно было назначить ближней явным образом с помощью описателя near. В соответствии с типом процедуры и команда ret, встретившаяся в этой процедуре, транслируется в код ближнего возврата C3h.

Из приведенного рассмотрения должно быть ясно, что ближние процедуры следует вызывать только из того же сегмента командой ближнего вызова call, в то время как процедуры, объявленные как дальние, следует вызывать только с помощью команды дальнего вызова call far ptr. Лишь в этом случае завершающие эти процедуры команды ret будут работать правильно.

Статья 36. **Косвенные вызовы подпрограмм**

До сих пор при вызове подпрограмм мы пользовались командой call с указанием в качестве ее операнда имени вызываемой подпрограммы. Такой вызов подпрограммы называется прямым; он нагляден, но не отличается гибкостью. Действительно, для того чтобы той же строкой вызвать другую подпрограмму, необходимо изменить исходный текст и перетранслировать программу. Большей гибкостью обладают косвенные вызовы, в которых адрес перехода извлекается не из кода команды, а из ячеек памяти или регистров; в коде команды содержится информация о том, где находится адрес перехода. Изменяя программным образом содержимое адресуемых командой call ячеек или регистров, т. е. засылая в них адрес той или иной подпрограммы, можно программно настроить команду call на вызов требуемой в настоящий момент подпрограммы.

В примере 36.1 дана иллюстрация косвенного вызова с использованием для адреса вызываемой подпрограммы ячейки памяти.

Пример 36.1. Косвенный вызов подпрограмм

```
;Главная процедура main
main    proc
        ...
;Выведем с помощью функции DOS 09h сообщение prompt
        ...
;Поставим запрос на ввод символа
inpt:   mov    AH,01h       ;Функция ввода символа с эхом
        int    21h
        cmp    AL,'y'       ;Нажата клавиша Y?
        je     mode_dos     ;Да, работаем с DOS
        cmp    AL,'n'       ;Нажата клавиша N?
        je     mode_bios    ;Да, работаем с BIOS
```

```
        jmp    inpt         ;Нажато не Y/N, повторить ввод
mode_dos:mov   addr,offset dos ;Зашлем адрес процедуры dos
        jmp    cont         ;и на продолжение
mode_bios:mov  addr,offset bios;Зашлем адрес процедуры bios
cont:   call   DS:addr      ;Косвенный вызов процедуры
;Завершим программу
        ...
main    endp
;Процедура вывода средствами DOS
dos     proc
        mov    AH,09h
        mov    DX,offset mes1
        int    21h
        ret
dos     endp
;Процедура вывода средствами BIOS
bios    proc                ;Процедура вывода средствами BIOS
;Очистим экран
        mov    AH,06h       ;Функция задания окна
        mov    AL,0         ;Режим создания окна
        mov    BH,07h       ;Атрибут всех символов в окне (ч/б)
        mov    CX,0         ;Координаты верхнего левого угла
        mov    DH,79        ;Нижняя Y-координата
        mov    DL,24        ;Правая X-координата
        int    10h          ;Прерывание BIOS
;Выведем строку
        mov    AH,13h       ;Функция вывода строки
        mov    AL,0         ;Режим 0 (атрибут в BL)
        mov    BH,0         ;Видеостраница
        mov    BL,0Eh       ;Атрибут всех символов
        mov    CX,len       ;Длина строки
        mov    DH,12        ;Начальная позиция, строка
        mov    DL,20        ;Начальная позиция, столбец
        push   DS           ;Настроим ES на наш
        pop    ES           ;сегмент данных
        mov    BP,offset mes2;ES:BP → выводимая строка
        int    10h          ;Прерывание BIOS
;Позиционируем курсор в начало последней строки экрана
        mov    AH,02h       ;Функция позиционирования
        mov    BH,0         ;Видеостраница
        mov    DH,24        ;Строка
        mov    DL,0         ;Столбец
        int    10h          ;Прерывание BIOS
        ret                 ;Возврат из прерывания
bios    endp
;Поля данных
addr    dw 0                ;Поле для адреса подпрограммы
prompt  db 'Драйвер ANSI.SYS установлен? [Y/N] : $'
mes1    db 27,'[2J',27,'[12;20H',27,'[31;1m'
        db 'Начинаем работать, используя средства DOS'
        db 27,'[0m',27,'[25;1H$'
mes2    db 'Начинаем работать, используя средства BIOS'
len=$-mes2
```

В программе имеются главная процедура main и две процедуры-подпрограммы dos и bios. Обе выполняют очистку экрана и вывод в середину экрана "своей" цветной строки "Начинаем работать...". Процедура dos использует для очистки экрана, позиционирования курсора и задания цвета Esc-последовательности и, значит, может функционировать только при наличии в системе драйвера ANSI.SYS. Процедура bios выво-

выводит на экран то же самое, но средствами BIOS и по этой причине не требует драйвера ANSI.SYS. В целях наглядности в процедурах для выводимых символов используются разные цвета.

Основная процедура выводит на экран вопрос о наличии в системе драйвера ANSI.SYS и вводит в программу ответ пользователя в виде символов Y (yes, да) или N (no, нет). Далее введенный символ анализируется и в зависимости от ответа пользователя осуществляется переход на метки mode_dos или mode_bios. В предложениях с этими метками ячейка addr загружается адресом требуемой подпрограммы. После настройки ячейки addr выполняется команда call DS:addr, которая и осуществляет косвенный переход.

Указание перед именем ячейки памяти обозначения сегментного регистра, в данном случае DS, задает косвенность вызова. Другой способ задания того же – описатель word ptr (word pointer, указатель на слово):

```
        call  word ptr addr
```

Косвенные вызовы можно выполнять с помощью широкого набора способов адресации. Так, если адрес вызываемой подпрограммы занести в один из базовых или индексных регистров, то команда вызова упрощается:

```
        mov   BX,offset bios ;В BX адрес самой подпрограммы
        call  BX             ;Косвенный вызов
```

Регистровая адресация возможна и в том случае, когда адрес подпрограммы находится в ячейке памяти:

```
        mov   SI,offset addr ;SI=адрес ячейки с адресом подпрогр.
        call  [SI]           ;Косвенный вызов
```

Легко сообразить, что использование в примере 36.1 косвенной адресации носит несколько искусственный характер. Можно было поступить проще и в зависимости от кода введенной команды выполнить одну из команд прямого вызова:

```
mode_dos:call  dos           ;Вызываем процедуру dos
        jmp   cont           ;и на продолжение
mode_bios:call bios          ;Вызываем процедуру bios
cont:                        ;Продолжение программы
```

Мы хотели только проиллюстрировать принцип использования косвенного вызова процедур. В дальнейших статьях книги будут приведены более оправданные примеры использования косвенных вызовов, которые не могут быть заменены прямыми.

Приведенный пример еще раз наглядно показывает характерные различия средств DOS и BIOS. Вывод на экран с помощью функций DOS отличается простотой (процедура dos состоит всего из четырех предложений, включая команду ret), но DOS обеспечивает лишь минимум возможностей. Для вывода цветного текста нам пришлось использовать "надстройку" над DOS – специальный драйвер ANSI.SYS с его Esc-последовательностями. Этот драйвер не всегда установлен в системе, да и применение Esc-последовательностей не очень удобно. Использование прерывания BIOS приводит к значительно более громоздкой программе (в процедуре bios 24 команды) при существенном увеличении возможностей вывода (позиционирование курсора, изменение цвета символов и пр.). Как уже отмечалось ранее, в реальных прикладных программах Esc-последовательности используются редко; обычно вывод на экран осуществляется с помощью средств BIOS или прямым обращением к видеопамяти.

Статья 37. **Прерывания пользователя**

Выше (см. статью 7) уже отмечалось значение прерываний для организации работы вычислительной системы. С помощью программных прерываний осуществляется доступ к системным средствам обслуживания аппаратуры компьютера и вычислительного процесса; аппаратные прерывания от периферийного оборудования компьютера позволяют процессору мгновенно реагировать на такие события, как срабатывание таймера, перемещение мыши или нажатие клавиши на клавиатуре; внутренние прерывания помогают обрабатывать ошибки вроде деления на нуль или отказа питания; свободные векторы (прерывания пользователя) применяется для взаимодействия компонентов сложных программных комплексов.

Использование механизма прерываний для вызова системных средств (функций DOS и BIOS) было рассмотрено в статьях 7 и 9. Поскольку при вызове системных средств обработчики соответствующих программных прерываний уже имеются в составе операционной системы, задача программиста сводится лишь к соответствующей настройке регистров и включению в программу команды int с требуемым номером. В тех же случаях, когда в прикладной программе необходимо обрабатывать прерывания от аппаратуры или программные прерывания пользователя, программисту приходится создавать собственные обработчики прерываний. Общие принципы построения обработчиков аппаратных и программных прерываний одинаковы, однако обработка аппаратных прерываний требует учета целого ряда дополнительных соображений. Рассмотрим сначала организацию программного комплекса с обработчиком программного прерывания пользователя.

С точки зрения конечного результата вызов из прикладной программы обработчика программного прерывания не отличается от вызова подпрограммы. И в том и в другом случае ход основной программы прерывается, управление передается на обработчик прерывания или подпрограмму, а после их завершения продолжается выполнение вызывающей программы. Различия заключаются лишь в способе вызова: подпрограмма вызывается командой call, а обработчик прерывания – командой int. Преимущества оформления отдельных фрагментов программ в виде обработчиков программных прерываний становятся очевидными при использовании резидентных программ, для которых программные прерывания являются стандартным способом взаимодействия с другими программами. Кроме того, обработчики программных прерываний широко используются в тех случаях, когда прикладной программе требуется внедриться в системный обработчик того или иного программного прерывания, например прерывания DOS int 21h или одного из прерываний BIOS. Все эти методики будут рассмотрены в дальнейшем; в настоящей статье приведен несколько искусственный пример обработчика прикладного программного прерывания, встроенного в обычную (нерезидентную) программу.

При отладке сложных программ часто возникает необходимость получить "мгновенный снимок" состояния программы в той или иной точке. Изучение содержимого регистров процессора позволяет установить, в какой области оперативной памяти находится и к какой области обращается программа, какой стек она использует, какие флаги установлены, и т. д. Далеко не всегда эту информацию можно получить с помощью отладчика, хотя бы потому, что программа, выполняемая под управлением отладчика, загружается не по тем адресам, где она будет работать в "самостоятельном" ре-

жиме. Кроме того, отладчик при запуске обнуляет регистры процессора и тем самым искажает операционную среду, в которой работает программа. Поэтому весьма полезным может оказаться "добавление" к отлаживаемой программе, позволяющее выводить на экран в заданных точках программы содержимое регистров процессора (или важных для выполнения программы участков памяти). Это "добавление" удобно оформить в виде обработчика какого-либо свободного программного прерывания. Тогда включение в программу в критических точках команд int позволит контролировать ход ее выполнения.

Для упрощения нашего примера мы ограничимся выводом на экран единственного числа – содержимого регистра флагов. Поскольку в регистре флагов, как и в любом другом регистре процессора, хранится двоичное число, для вывода на экран его следует предварительно преобразовать в символьную форму. Этот вопрос уже был рассмотрен в статье 13; в настоящем примере мы воспользуемся подпрограммами преобразования bin_asc и wrd_asc из этой статьи.

Для того чтобы некоторый программный фрагмент вызывался командой int *n*, надо просто поместить в выбранный вектор с номером *n* адрес этого фрагмента, который должен завершаться командой iret. Тогда команда int *n* в отлаживаемой программе будет передавать управление обработчику этого прерывания, а команда iret, завершающая обработчик, будет возвращать управление в отлаживаемую программу. Естественно, выбранный вектор не должен использоваться системой и загруженными или запускаемыми прикладными программами.

Если с помощью какой-либо инструментальной программы (например, Manifest фирмы Quarterdeck) просмотреть таблицу векторов, то можно заметить, что значительная их часть не используется. Сюда относятся, например, векторы 60h...66h, которые специально отведены для использования в прикладных программах, а также векторы 32h, 34h...3Fh, 42h, 78h...7Fh и ряд других, зарезервированных для дальнейшего использования системой. Любым из этих векторов в принципе можно воспользоваться "в личных целях". При этом, однако, следует иметь в виду, что коммерческие программы, устанавливаемые на компьютере, часто используют свободные векторы. Например, резидентная антивирусная программа PCCSTSR.COM фирмы Trend Micro Devices применяет вектор пользователя 60h. Поэтому, создавая свою резидентную программу, активизируемую через вектор прерывания, надо предварительно убедиться, что выбранный вектор действительно свободен.

Команда int *n* преобразуется в результате трансляции в код CD *n* и занимает 2 байта. Специально для отладочных целей в системе команд микропроцессора предусмотрена команда int 3, которая занимает всего 1 байт. В ее машинный код (CCh) не входит номер вектора, однако она всегда вызывает программу, адрес которой находится в векторе 3. Следует только иметь в виду, что интерактивные отладчики сами используют вектор 3 в служебных целях. Поэтому проанализировать работу приведенного ниже примера в стандартном отладчике не удастся – он будет конфликтовать с программой отладчика. Можно порекомендовать с целью отладки сначала использовать в программе какой-либо из свободных векторов, например 61h или 62h, а затем заменить его на вектор 3.

Пример 37.1. Обработчик программного прерывания 03h

```
.586                            ;(1)
text      segment use16         ;(2)
          assume  CS:text,DS:data;(3)
```

```
;Прикладной обработчик прерывания 03h
new_03h proc              ;(4)
        pusha             ;(5)Сохраним все регистры
        push  DS          ;(6)И еще DS
        push  CS          ;(7)Настроим DS
        pop   DS          ;(8)на сегмент команд
        mov   BP,SP       ;(9)SP=текущая вершина стека
        mov   AX,[BP+22]  ;(10)AX=регистр флагов до int
        mov   SI,offset flags+6;(11)CS:SI=поле для результата
        call  wrd_asc     ;(12)Преобразуем AX (флаги)
        mov   AH,09h      ;(13)Функция DOS вывода на экран
        mov   DX,offset flags;(14)DS:DX=flags
        int   21h         ;(15)Вызов DOS
        pop   DS          ;(16)Восстановим DS
        popa              ;(17)Восстановим все регистры
        iret              ;(18)Возврат в вызывающую программу
flags   db 'FLAGS=****h $';(19)Поле для данных в сегменте команд
new_03h endp              ;(20)
;Главная процедура
main    proc              ;(21)
        mov   AX,data     ;(22)Стандартные действия по
        mov   DS,AX       ;(23)инициализации DS
;Установим прикладной обработчик
        mov   AX,3503h    ;(24)Чтение исходного содержимого
        int   21h         ;(25)вектора 03h
        mov   word ptr old_03h,BX;(26)Сохраним исходный
        mov   word ptr old_03h+2,ES;(27)вектор
        mov   AX,2503h    ;(28)Заполнение вектора 03h
        mov   DX,offset new_03h;(29)DX=смещение обработчика
        push  DS          ;(30)Сохраним на время DS
        mov   BX,seg new_03h;(31)BX=сегмент обработчика
        mov   DS,BX       ;(32)DS=сегмент обработчика
        int   21h         ;(33)Вызов DOS - заполнение вектора
        pop   DS          ;(34)Восстановим DS
;Далее может следовать текст отлаживаемой программы,
;в которую в критических точках включены команды int 3.
;У нас этой программы нет, мы ограничимся командой int 3
        int   3           ;(35)Вывод на экран флагов процессора
;Перед завершением программы восстановим вектор
        mov   AX,2503h    ;(36)Функция записи вектора
        lds   DX,old_03h  ;(37)DS:DX=исходное содержимое вектора
        int   21h         ;(38)Вызов DOS
;Завершим программу
        mov   AX,4C00h    ;(39)
        int   21h         ;(40)
main    endp              ;(41)
;Подпрограмма преобразования слова в символьную форму
wrd_asc proc
        ...
wrd_asc endp
;Подпрограмма преобразования четверки битов в символ
bin_asc proc
        ...
bin_asc endp
text    ends
data    segment use16
old_03h dd 0  ;Ячейка для хранения исходного вектора
data    ends
stk     segment stack
        db    256 dup (0) ;Стек
stk     ends
```

```
end   main
```

Исходный текст примера 37.1 состоит из целого ряда программных фрагментов: обработчика прерывания, главной процедуры, подпрограмм wrd_asc и bin_asc. Как уже отмечалось ранее, взаимное расположение фрагментов программы не имеет ни малейшего значения. Можно было начать текст программы с главной процедуры или, например, с подпрограмм. Существенно, однако, чтобы все подпрограммы завершались командами ret, обработчики прерываний – командами iret, а главная процедура оканчивалась вызовом функции DOS 4Ch. Необходимо также указать в качестве параметра завершающей команды end смещение точки входа в программу, т. е. смещение главной процедуры.

Установка прикладного обработчика прерываний в большинстве случаев состоит из двух шагов. Прежде всего в выделенной для этого двухсловной ячейке памяти (old_03h в примере 28.1) сохраняется исходное содержимое используемого вектора, чтобы перед завершением программы можно было вернуть таблицу векторов в исходное состояние. Прочитать вектор можно прямым обращением к памяти (вектор 3 занимает ячейки с адресами 0000h:000Ch...0000h:000Fh), однако лучше использовать специально предназначенную для этого функцию DOS с номером 35h (предложения 24 и 25). При ее вызове в регистр AL заносится номер интересующего нас вектора; содержимое вектора возвращается в регистрах ES:BX (естественно, в ES сегментный адрес из вектора, а в BX – смещение).

Сохранив вектор, следует занести в него адрес нашего обработчика, для чего используется функция DOS с номером 25h. Как и в случае функции 35h, номер вектора указывается в регистре AL; полный двухсловный адрес обработчика должен находиться в регистрах DS:DX. Здесь нас подстерегает неприятность: регистр DS обычно настраивается на сегмент данных, однако к моменту вызова функции 25h в нем должен содержаться сегментный адрес того сегмента команд, в котором находится обработчик прерываний. Поэтому мы сохраняем на время исходное содержимое DS (предложение 30), получаем сегментный адрес обработчика (предложение 31) и заносим его в DS, после чего можно обратиться к DOS (предложение 33), которая и заполнит вектор прерывания. В нашем случае сегмент обработчика совпадает с сегментом программы, адрес которого находится в CS, и перенести его в DS можно было через стек:

```
push   CS
pop    DS
```

Приведенный выше фрагмент эффективнее использованного в тексте программы. Восстановлением сохраненного на время содержимого DS завершается этап инициализации обработчика прерывания.

Описанная выше секция инициализации должна, разумеется, располагаться перед текстом отлаживаемой программы (или, во всяком случае, перед текстом ее отлаживаемого участка). Включение в любом месте следующей далее программы команды int 3 приведет к выводу на экран содержимого регистра флагов в этой точке.

Перед завершением программы необходимо с помощью функции DOS 25h восстановить измененный нами вектор прерывания. Перенести в регистры DS:DX содержимое ячейки old_03h можно двумя командами mov, однако удобнее воспользоваться командой lds (предложение 37), которая в одном действии заполняет сразу оба регистра.

Перейдем теперь к рассмотрению самого обработчика прерываний (процедура new_03h, предложение 4). Необходимо помнить, что в обработчике прерываний, как и

в подпрограмме, недопустимо разрушение регистров, так как это может нарушить ход выполнения вызывающей программы. Поэтому в начале обработчика необходимо сохранить в стеке все регистры, которые в нем будут использоваться, а в конце восстановить их. Мы для простоты воспользовались командой pusha сохранения всех регистров. Однако эта команда сохраняет только регистры данных, нам же понадобится еще и регистр DS, который приходится сохранять отдельной командой (предложение 6). В следующих далее предложениях 7 и 8 регистр DS настраивается на сегмент обработчика. Зачем это нужно?

Наш обработчик прерывания использует "инструментальное" поле данных flags (предложение 19), в которое при вызове подпрограммы wrd_asc заносится символьное представление содержимого регистра флагов. Это поле можно было расположить в сегменте данных программы, однако, поскольку оно носит инструментальный характер и нужно только обработчику, разумнее поместить его в процедуру обработчика. Разумеется, это поле не должно разбивать команды обработчика, однако после команды iret оно вполне уместно. Содержимое поля flags будет выводиться на экран функцией DOS 09h, которая требует наличия адреса выводимой строки в регистрах DS:DX. Если мы хотим воспользоваться функцией 09h, мы обязаны настроить DS на тот сегмент, в котором находится выводимое сообщение.

Есть и еще одно, более тонкое соображение. В подпрограмме wrd_asc для заполнения поля flags используется индексная адресация через регистр SI:

```
mov    byte ptr [SI],AL
```

Как уже неоднократно отмечалось, при любом обращении к памяти процессор использует два компонента адреса. Смещение в нашем случае находится в регистре SI, а сегментный регистр в приведенной выше команде не указан. Однако по умолчанию такого рода команда транслируется так, что процессор при ее выполнении использует в качестве сегментного регистр DS. Таким образом, и эта команда требует помещения в DS сегментного адреса сегмента команд. Если бы в программе не использовалась функция DOS, которая тоже работает с DS, а была бы лишь команда обращения к ячейке памяти flags, находящейся в сегменте команд, то можно было бы обойтись без настройки DS, а команду обращения к памяти записать таким образом:

```
mov    byte ptr CS:[SI],AL
```

В этом случае адресация памяти будет осуществляться через регистр CS, который, разумеется, и так настроен на сегмент команд.

Уместно еще раз подчеркнуть, что неправильное содержимое сегментных регистров не может быть обнаружено на этапе трансляции, а проявляется лишь при выполнении программы. Поэтому программист должен очень внимательно относиться ко всем командам с обращением к памяти и заботиться о правильном заполнении используемых этими командами сегментных регистров.

Две следующие команды обработчика (предложения 9 и 10) иллюстрируют широко распространенную методику работы со стеком. По условию задачи нам надо получить содержимое регистра флагов в точке вызова обработчика в главной процедуре. Это содержимое было сохранено в стеке процессором при выполнении команды int (см. статью 7). Где оно там находится? В каждой конкретной программе адрес нужного места в стеке приходится определять индивидуально. В нашем случае состояние стека к моменту выполнения предложения 8 приведено на рис. 37.1.



Рис. 37.1. Состояние стека в обработчике прерывания примера 37.1

Команда int помещает в стек три слова – флаги, CS и IP. Первая команда обработчика pusha сохраняет в стеке содержимое восьми регистров в том порядке, какой указан на рис. 37.1. Следующая далее команда push DS смещает стек еще на одно слово. Таким образом, нужное нам слово флагов находится на 11 слов, или 22 байт, ниже вершины стека. Для чтения этого слова текущее содержимое SP переносится в BP и командой

```
mov     AX,[BP+22]
```

выполняется загрузка в регистр AX слова флагов. Напомним, что при использовании в качестве индексного регистра BP процессор по умолчанию обращается к области памяти, адресуемой через SS, т. е. к стеку. Рассмотренная методика работы со стеком, когда сначала регистр BP настраивается на текущую вершину стека, а затем обращение к стеку осуществляется с помощью индексной адресации через BP с указанием дополнительного числового смещения, является общепринятой.

Поместив в регистр AX интересующее нас данное, мы настраиваем SI на адрес поля для размещения результирующей символьной строки, вызываем подпрограмму word_asc преобразования числа в символы и выводим эту строку на экран функцией DOS 09h. В конце обработчика восстанавливаются сохраненные ранее регистры, после чего обработчик завершается командой выхода из прерывания iret.

Запустив программу 37.1, вы получите на экране строку вроде следующей:

```
FLAGS=3202h
```

Она говорит о том, что в регистре флагов были установлены разряды 13, 12, 9 и 1 (см. рис. 3.2). Разряды 13 и 12 (уровень привилегий ввода-вывода IOPL) в реальном режиме не используются, и мы о них пока говорить не будем, разряд 9 соответствует установленному флагу IF, а разряд 1 в регистре флагов попросту отсутствует, но при чтении всегда дает 1. Конкретное значение регистра флагов, выводимое на экран, может различаться в зависимости от операционной среды – запускается ли программа из Windows в сеансе командной строки или непосредственно из MS-DOS.

При необходимости программу можно сделать более универсальной, предусмотрев в ней вывод всех регистров процессора. Для этого придется удлинить символьную строку

flags, отведя в ней место для символьного представления всех выводимых регистров, и, перенося по очереди содержимое остальных регистров в AX, вызывать для каждого подпрограмму wrd_asc, не забывая при этом модифицировать смещение в SI. При этом исходное содержимое регистров DS, SI и BP, которые используются и модифицируются в процедуре new_03h, придется извлечь из стека (как это было сделано для регистра флагов), определив по рис. 37.1 их смещения. То же касается и исходного содержимого указателя команд IP, которое вообще нельзя получить другим способом, так как IP не является мнемоническим обозначением указателя команд и предложение вида

```
        mov    AX,IP
```

не имеет смысла.

Статья 38. **Обработка аппаратных прерываний**

В статье 27 была рассмотрена работа системного таймера. Там же были приведены некоторые примеры программирования таймера, в частности процедура изменения его частоты, однако не затрагивались вопросы обработки прерываний от таймера. В действительности же таймер используется главным образом именно как источник периодических прерываний. Напомним, что при настройке по умолчанию (в регистре-фиксаторе содержится число FFFFh) на выходе таймера возникают сигналы с частотой 1,19318 МГц : 65 535=18,2 Гц. Эти сигналы возбуждают прерывания на линии IRQ0 с вектором 08h, которые в нормальном режиме работы компьютера обрабатываются программой BIOS, осуществляющей отсчет текущего времени. Прерывания канала 0 широко используются для временной синхронизации прикладных программ и программно-управляемых процессов. Например, с помощью прерываний от таймера можно периодически выводить на экран некоторую информацию, отсчитывать интервалы времени, управлять в реальном времени аппаратурой, подключаемой к компьютеру, и т. д. Рассмотрим простой обработчик прерываний от системного таймера.

В примере 38.1 задачей обработчика прерываний от таймера является отсчет заданного интервала времени (конкретно – 3 с) и по истечении этого интервала переключение основной программы на другую ветвь ее выполнения. В результате программа в течение 3 с выполняет одну работу, а по истечении 3 с переключается на выполнение других действий. В рассматриваемом примере программа в течение 3 с просто крутится в цикле, а через 3 с выходит из этого цикла и, выведя на экран предупреждающее сообщение, завершается. В реальном случае программа могла бы, включив некоторую измерительную аппаратуру, в течение заданного интервала времени выводить на экран информацию, получаемую из этой аппаратуры, а по истечении интервала времени выключить аппаратуру и приступить к обработке накопленных данных.

Измерение интервала времени осуществляется путем счета числа поступивших прерываний. Интервал 3 с соответствует приблизительно 54 прерываниям; соответственно на первые 53 прерывания обработчик прерываний просто завершается, а на 54-е выполняет переключение ветвей основной программы.

Пример 38.1. Обработчик прерываний от таймера

```
main    proc
        mov    AX,data        ;Инициализация
        mov    DS,AX          ;сегментного регистра DS
;Прочитаем и сохраним исходное содержимое вектора 8
        mov    AX,3508h
```

```
        int     21h
        mov     word ptr old_08h,BX
        mov     word ptr old_08h+2,ES
;Установим наш обработчик прерываний new_08h
        mov     AX,2508h
        mov     DX,offset new_08h
        push    DS              ;Сохраним на время DS
        push    CS              ;Отправим содержимое CS
        pop     DS              ;в DS
        int     21h             ;Вызов DOS (функции 25h)
        pop     DS              ;Восстановим DS
;Сымитируем действия, выполняемые в течение 3 с, просто зациклив программу
stop:   jmp     stop
;Вторая ветвь программы, активизируемая по истечении 3 с
fin:    mov     AH,09h          ;Выведем на экран сообщение
        mov     DX,offset msg
        int     21h
        mov     AX,2508h        ;Восстановим вектор 8
        lds     DX,old_08h
        int     21h
        mov     AX,4C00h        ;Завершим программу
        int     21h
main    endp
;Прикладной обработчик прерываний от таймера,
;активизируемый 18,2 раза в секунду
new_08h proc
        push    AX              ;Сохраним два используемых
        push    BP              ;в обработчике регистра
        dec     CS:time         ;Декремент интервала времени
        jnz     outint          ;Пока не 0, выйти из прерывания
;Содержимое ячейки time уменьшилось до 0, выполнить переключение программы
        mov     BP,SP           ;BP=текущая вершина стека
        mov     AX,offset fin;Смещение точки перехода
        mov     [BP+4],AX       ;Отправим его в стек на место IP
        mov     AX,seg fin      ;Сегмент точки перехода
        mov     [BP+6],AX       ;Отправим его в стек на место CS
outint: mov     AL,20h          ;Команда EOI в контроллер
        out     20h,AL          ;прерываний
        pop     BP              ;Восстановим оба
        pop     AX              ;сохраненных регистра
        iret                    ;Выход из прерывания
time    dw 54                   ;Ячейка для отсчета времени
new_08h endp
;Поля данных
old_08h dd 0                    ;Ячейка для хранения исходного вектора
msg     db 'Временной интервал истек!$';Предупреждающее сообщение
```

Процедура установки обработчика аппаратного прерывания ничем не отличается от той, что была рассмотрена в статье 37, посвященной программным прерываниям: сначала с помощью функции 35h читается и сохраняется в ячейке old_08h исходное содержимое вектора 8, затем функцией 25h в вектор заносится полный адрес прикладного обработчика new_08h. После этого программа приступает к действиям, запланированным для выполнения в течение 3-секундного интервала (в данном случае – цикл перехода на метку stop).

Процедура new_08h обработчика прерываний весьма проста. После сохранения в стеке регистров AX и BP, используемых в программе обработчика, выполняется декремент ячейки time, исходное содержимое которой определяет измеряемый интервал времени. Эта ячейка расположена фактически в процедуре обработчика и обращение к

ней требует замены сегмента (DS на CS). Если результат не достиг нуля, командой jnz осуществляется переход на метку outint и завершение обработки данного прерывания. В процедуру завершения входит генерация команды EOI, посылаемой в ведущий контроллер прерываний (см. статью 26), восстановление сохраненных ранее регистров и команда выхода из прерывания iret.

При обработке 54-го прерывания команда dec time фиксирует нулевой результат, в результате чего команда условного перехода jnz не срабатывает и выполняются следующие строки обработчика. Регистр BP настраивается на текущую вершину стека, и на место находящегося в стеке двухсловного адреса возврата в основную программу записывается двухсловный адрес ячейки fin, на которую мы хотим переключить основную программу по истечении 3 с. Смещение на 4 байта относительно вершины стека понадобилось из-за того, что в начале программы обработчика мы сохранили в стеке содержимое двух регистров.

Рассмотренный способ динамического изменения адреса возврата путем модификации содержимого стека используется иногда и в прикладном, и в системном программировании, однако главным образом в обработчиках программных (синхронных) прерываний. Реализация такого приема в обработчике аппаратных (асинхронных) прерываний, как это сделано в нашем примере, приводит к невозможности использовать в основной программе системные средства. Дело в том, что диспетчер DOS, вызываемый командой int 21h, выполняет, в числе прочего, переключение стека: вместо стека прикладной программы начинает использоваться стек DOS. В этом случае при поступлении прерывания от таймера процессор сохраняет в стеке в качестве адреса возврата текущий адрес прерванной программы DOS. Ясно, что, заменив его адресом точки stop нашей программы, мы после выхода из прерывания уже не вернемся в DOS, работа которой по выполнению затребованной функции останется незавершенной. Это приведет к выходу системы из строя и необходимости перезагрузки машины. Именно поэтому в примере 38.1 останов программы в ожидании прихода 54-го прерывания от таймера выполнен с помощью бесконечного цикла

```
stop:   jmp   stop
```

а не путем вызова, например, функции 01h DOS.

Статья 39. **Взаимодействие прикладных и системных обработчиков прерываний**

Методика полной подмены системного обработчика прикладным, использованная в программе предыдущей статьи (так называемый перехват вектора), обладает существенным недостатком: на время функционирования программы отключается системное обслуживание прерываний от таймера, что приводит к останову системных часов. В этом легко убедиться, увеличив в примере 38.1 интервал времени до 10-15 с и выполнив командный файл, состоящий из такой последовательности команд DOS:

```
TIME
30-01.EXE
TIME
```

Сравнение выводов команд TIME до запуска нашей программы и после ее завершения покажет, что системное время практически не увеличилось. Следовательно,

системные часы стояли, пока выполнялась наша программа. Следует, однако, заметить, что этот эксперимент покажет убедительные результаты, лишь если он выполняется в чистой DOS. Если повторить его в сеансе DOS системы Windows 95 (или Windows 3.1), то показания системных часов после завершения программы будут соответствовать правильному астрономическому времени; отставания часов мы не обнаружим. Объясняется это явление тем, что, хотя программа, перехватывая прерывания от таймера, останавливает отсчет системного времени, Windows после завершения программы тут же восстанавливает для данного сеанса DOS правильное время. Этот пример показывает, как осторожно следует относиться к запуску в системе Windows приложений DOS, использующих временную синхронизацию событий. Системы Windows NT/2000 имеют существенные отличия от Window 95/98, и в них наша программа будет работать так же, как и в чистой DOS, – на время ее работы системный отсчет времени будет остановлен, что приведет к отставанию системных часов.

Между тем существует такой способ организации прикладного обработчика, при котором системные алгоритмы обслуживания прерывания не отключаются, а продолжают действовать как обычно. Иногда этот способ называют сцеплением обработчиков или реализацией цепочки обработчиков прерывания. Он применим как к аппаратным, так и к программным прерываниям.

При инициализации прикладного обработчика, сцепляемого с системным, следует точно так же, как и при полном перехвате вектора, сохранить адрес системного обработчика (для общности назовем двухсловную ячейку для хранения исходного вектора old_vec) и поместить в вектор прерывания адрес (например, new_int) прикладного обработчика. Для обеспечения выполнения при каждом прерывании сначала системной обработки, а за ней прикладной структура прикладного обработчика должна быть такой:

```
new_int proc
        pushf                   ;Сохраним флаги для команды iret
        call   dword ptr old_vec;В системный обработчик с возвратом
        ...                     ;Прикладная обработка
        iret
new_int endp
```

После того как процессор выполнит процедуру прерывания, в стеке прерванного процесса оказываются три слова: слово флагов, а также двухсловный адрес возврата в прерванную программу (рис. 39.1). Именно такая структура данных должна быть на верху стека, чтобы команда iret, которой завершается любая программа обработки прерываний, могла вернуть управление в прерванный процесс.

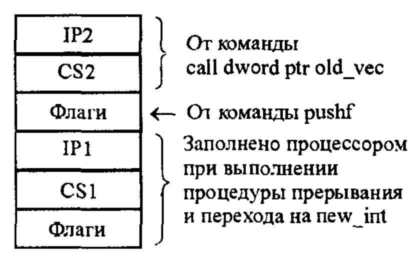


Рис. 39.1. Стек прерванной программы в процессе выполнения прикладного обработчика прерываний: CS1:IP1 - адрес точки возврата в прерванную программу, CS2:IP2 – адрес точки возврата в прикладной обработчик

Первая команда нашего обработчика pushf засылает в стек еще раз слово флагов, а команда дальнего косвенного вызова системного обработчика call dword ptr old_vec в процессе передачи управления системному обработчику помещает в стек двухсловный адрес возврата на следующую команду прикладного обработчика. В результате осуществляется переход на программу системного обработчика, а в стеке формируется трехсловная структура, необходимая для команды iret.

Системный обработчик, закончив обработку данного прерывания, завершается командой iret. Эта команда забирает из стека три верхних слова и осуществляет переход по адресу CS2:IP2, т. е. на продолжение прикладного обработчика. Завершающая команда нашего обработчика iret снимает со стека три верхних слова и передает управление по адресу CS1:IP1 в прерванную программу.

Иногда нужно обеспечить условия для выполнения прикладной обработки не после, а до системной. Тогда структура прикладного обработчика будет иной:

```
new_int proc
...                       ;Прикладная обработка
        jmp   dword ptr old_vec;В системный обработчик без возврата
new_int endp
```

Здесь нет необходимости помещать в стек флаги. Завершающая прикладную обработку команда перехода передает управление (не затрагивая стек) в системный обработчик, который далее выполняется обычным образом.

В редких случаях прикладной обработчик должен выполнить некоторые действия до передачи управления в системный, а некоторые – после. Тогда используется следующая структура обработчика:

```
new_int proc
        ...               ;Прикладная обработка до системной
        pushf             ;Сохраним флаги для команды iret
        call  dword ptr old_vec;В системный обработчик с возвратом
        ...               ;Прикладная обработка после системной
        iret
new_int endp
```

В практическом программировании в подавляющем большинстве случаев используется первый способ сцепления обработчиков, когда передача управления системному обработчику осуществляется командой call, а прикладная обработка выполняется после завершения системной.

Преобразуем программу 38.1, сцепив прикладной обработчик с системным. Это потребует лишь незначительного изменения первых строк процедуры обработчика; весь остальной текст программы остается без изменений (пример 39.1).

Пример 39.1. Прикладной обработчик прерываний от таймера, сцепленный с системным

```
new_08h proc
        pushf             ;Сохраним флаги для команды iret
        call  dword ptr old_08h
        push  AX          ;Сохраним два используемых
        push  BP          ;в обработчике регистра
```

Если выполнить (в чистой DOS) приведенный выше командный файл для программы 39.1, мы не обнаружим отставания системных часов. Таким образом, сцепление прикладного обработчика с системным позволяет, не нарушая работы компьютера, ввести в обработку прерываний штатных устройств прикладные алгоритмы.

Последний пример позволяет нам остановиться на весьма важном вопросе взаимодействия обработчика аппаратного прерывания и основной программы, в состав которой он входит. Как уже говорилось, в процессе обслуживания прерывания процессор изменяет содержимое лишь двух регистров – CS и IP, помещая в них адрес обработчика, извлеченный из таблицы векторов. Содержимое всех остальных регистров остается без изменения, в частности регистры SS:SP определяют вершину стека прерванной программы, а регистр DS указывает на ее сегмент данных. Это как будто предоставляет весьма привлекательную возможность записывать данные из обработчика прерываний непосредственно в основную программу. Например, обработчик прерываний от измерительной аппаратуры может принимать из нее результаты измерений, а основная программа тут же их обрабатывать. Однако в действительности это совсем не так. Аппаратные прерывания могут возникать в любой момент времени и прерывать основную программу в любой ее точке, и для того, чтобы обработчик прерываний мог обращаться к сегменту данных основной программы, требуется, чтобы содержимое регистра DS не изменялось в течение всей жизни программы. В реальных программах сегментные регистры данных приходится по ходу программы перенастраивать, например для того, чтобы обратиться к системным областям или к другим сегментам данных. В частности, многие системные вызовы в процессе своего выполнения настраивают регистр DS на системные поля данных. В результате попытка записи данных обработчиком прерываний в основную программу с большой вероятностью приведет к затиранию полей DOS и разрушению системы.

В свете сказанного ясно, что включенное в пример 39.1 предложение вызова системного обработчика через ячейку old_ptr, расположенную в сегменте данных основной программы

```
        call  dword ptr old_08h
```

выглядит не очень корректно. Поскольку в этой команде не указан сегментный регистр, по умолчанию процессор использует регистр DS, а его содержимое в процессе выполнения основной программы может измениться. Поэтому правильнее расположить ячейку old_08h с исходным вектором не в сегменте данных, а непосредственно в процедуре обработчика и обращаться к ней через регистр CS с использованием операции замены сегмента:

```
        call  CS:old_08h
```

Перенос поля данных в сегмент команд потребует изменения всех предложений, в которых происходит обращение к этой ячейке. Модифицированные фрагменты программы приведены ниже. Следует обратить внимание на расположение ячеек с данными. Их следует размещать так, чтобы ни при каких обстоятельствах процессор не начал бы выполнять коды данных. В частности, их можно поместить перед процедурой new_08h; мы нашли для них другое допустимое место – в самом конце этой процедуры, после команды iret (пример 39.2).

Пример 39.2. Сцепление прикладного обработчика с системным с расположением ячейки с исходным вектором в сегменте команд

```
main    proc
        mov   AX,data
        mov   DS,AX
        mov   AX,3508h
        int   21h
        mov   word ptr CS:old_08h,BX;Обращение к сегменту команд
```

```
        mov   word ptr CS:old_08h+2,ES;То же
        mov   AX,2508h
        mov   DX,offset new_08h
        ...
fin:    mov   AH,09h
        mov   DX,offset msg
        int   21h
;Восстановим исходное содержимое вектора 8
        mov   AX,2508h
        mov   DX,word ptr CS:old_08h;Смещение вектора в DX
        mov   BX,word ptr CS:old_08h+2;Сегмент вектора
        mov   DS,BX        ;перенесем в DS
        int   21h
        mov   AX,4C00h
        int   21h
main    endp
new_08h proc
        pushf
        call  CS:old_08h
        ...
        iret
time    dw 54              ;Ячейки с данными в сегменте
old_08h dd 0               ;команд
new_08h endp
text    ends
```

Статья 40. **Обработка прерываний по Ctrl+C и Ctrl+Break**

Во многих вычислительных системах сочетание клавиш Ctrl+C зарезервировано для принудительного завершения активной программы и передачи управления системе. Однако для этого нужно, чтобы DOS, обрабатывая прерывания от клавиатуры, постоянно анализировала поступающие коды и "вылавливала" код нажатия Ctrl+C (код ASCII 03h). MS-DOS проверяет наличие Ctrl+C во входном потоке не в программе обработки прерываний от клавиатуры, а на более высоком уровне, при выполнении программных запросов. При этом различные функции DOS по-разному реагируют на ввод с клавиатуры Ctrl+C.

Все функции DOS делятся на две группы – функции ввода-вывода с номерами 01h...0Ch и все остальные функции, т. е. функции с номерами 00h, 0Dh...6Ch, которые иногда называют "дисковыми". Функции с номерами, превышающими 6Ch, используются расширителями DOS, сетевыми программами, инструментальными пакетами и другими системными программами. Различие двух указанных групп заключается в том, что при вызове функций ввода-вывода DOS переходит на внутренний стек ввода-вывода, а при вызове всех остальных функций – на другой внутренний стек, который называется дисковым. Наличие в DOS двух внутренних стеков обеспечивает ее частичную реентерабельность (повторную входимость), т. е. возможность при обнаружении ошибки в процессе выполнения какой-либо функции DOS (принадлежащей к "дисковой" группе) вызвать функции ввода-вывода для вывода на экран аварийного сообщения и ввода с клавиатуры указаний пользователя. Вопросы нереентерабельности DOS и методы ее преодоления будут рассмотрены в статье 44.

Большая часть функций ввода-вывода из диапазона 01h...0Ch проверяют перед своим выполнением наличие в кольцевом буфере клавиатуры кода 03 (Ctrl+C) и при

обнаружении этого кода выполняют команду int 23h. В векторе 23h обычно находится адрес программы DOS, завершающей текущий процесс. Исключение составляют функции 06h и 07h, нечувствительные к Ctrl+C, а также функции 02h и 09h, которые (во всяком случае, в MS-DOS версий 5.0, 6.0 и 6.2) анализируют кольцевой буфер на предмет наличия там Ctrl+C один раз на каждые 64 вызова. Проверка на Ctrl+C осуществляется независимо от перенаправления ввода-вывода, а также независимо от состояния системного флага BREAK.

"Дисковые" функции выполняют проверку на Ctrl+C лишь в том случае, если установлен флаг BREAK, т. е. была выполнена команда DOS

```
BREAK ON
```

Таким образом, изменяя состояние BREAK, можно включать или выключать механизм реакции на Ctrl+C большинства функций DOS. Заметим, что речь идет практически обо всех функциях DOS: файловых, получения или установки даты и времени, выделения и освобождения памяти, запуска и завершения задач и др.

Поскольку проверка на Ctrl+C осуществляется только при выполнении функций DOS, нажатием Ctrl+C в системе MS-DOS нельзя завершить чисто процессорную (вычислительную) задачу, а только такую, в которой имеются вызовы системных функций.

Однако и такая возможность часто оказывается чрезмерной. При завершении задачи по Ctrl+C могут остаться невосстановленными модифицированные векторы прерываний или неправильно закрыться открытые файлы. Поэтому большинство прикладных программ не использует системный обработчик Ctrl+C, заменяя его собственным. Назначение этого обработчика – корректное завершение программы (вывод на экран запроса на подтверждение завершения, закрытие файлов, возможно, выключение оборудования, подсоединенного к компьютеру, и т. д.). Поскольку в момент передачи управления через вектор 23h система находится в обычном стабильном состоянии, это позволяет использовать в обработчике любые функции DOS (например, функции вывода). Как правило, обработчик завершается вызовом функции DOS 4Ch, передающей управление командному процессору COMMAND.COM.

При замене системного обработчика прикладным необходимо иметь в виду, что вектор 23h, как, впрочем, и любой другой, принадлежит не конкретной задаче, а всей вычислительной системе. После завершения задачи необходимо восстановить исходное содержимое вектора, так как в противном случае ввод Ctrl+C при выполнении последующих задач неминуемо приведет к нарушению работы системы. Однако DOS при загрузке задачи в память копирует в определенные ячейки префикса программы PSP содержимое векторов 22h (завершение задачи), 23h (обработка Ctrl+C) и 24h (обработка критической ошибки) (см. табл. 32.1). Стандартная системная процедура завершения задачи включает в себя восстановление исходного содержимого указанных трех векторов, которое берется из префикса программы. Таким образом, даже если прикладная программа, модифицировав вектор 23h, не восстановила его, это сделает DOS в процессе завершения задачи.

В примере 40.1 рассматривается простейший обработчик прерывания 23h, который, выведя на экран предупреждающее сообщение и дождавшись ответа пользователя (нажатия любой клавиши), завершает программу обычным образом – вызовом функции DOS 4Ch.

Пример 40.1. Обработчик Ctrl+C. Аварийное завершение программы

```
main    proc
```

```
;Занесем в вектор 23h адрес нашего обработчика
        mov    AX,2523h
        mov    DX,offset new_23h
        push   CS              ;Настроим DS на сегмент обработчика
        pop    DS              ;(требуется для функции 25h)
        int    21h
        mov    AX,data         ;Сделаем наш сегмент данных
        mov    DS,AX           ;адресуемым
        mov    AH,02h          ;Функция вывода на экран символа
        mov    DL,'*'          ;Выводимый символ
kkkk:   int    21h
        jmp    kkkk            ;Бесконечный цикл вывода символа
main    endp
;Прикладной обработчик прерывания 23h
new_23h proc
;Выведем информационное сообщение
        mov    AH,09h
        mov    DX,offset mesg
        int    21h
;Завершим программу обычным образом
        mov    AX,4C00h
        int    21h
new_23h endp
;В сегменте данных
mesg    db     'Хотите ли вы завершить программу?$'
```

В главной процедуре программы main выполняется загрузка в вектор 23h адреса прикладного обработчика. Поскольку после завершения программы исходное содержимое этого вектора будет восстановлено системой, нет необходимости в его предварительном сохранении.

После инициализации вектора 23h программа входит в бесконечный цикл вывода на экран символа. При нажатии сочетания Ctrl+C управление передается на прикладной обработчик, который выводит на экран сообщение mesg и завершает программу вызовом функции DOS 4Ch.

Система MS-DOS предоставляет и другую возможность вмешательства в ход выполнения программы – нажатие клавиш Ctrl+Break.

Системный обработчик прерываний от клавиатуры, входящий в состав BIOS, при обнаружении комбинации клавиш Ctrl+Break передает управление программе, адрес которой содержится в векторе 1Bh. Эта программа, так же входящая в состав BIOS, состоит из единственной команды iret и не выполняет, таким образом, никаких функций. Однако в процессе начальной загрузки DOS изменяет содержимое вектора 1Bh, записывая в него адрес своего обработчика. Этот обработчик, получив управление, выполняет следующие действия:

- включает 0000h в кольцевой буфер клавиатуры на место головного символа;
- модифицирует указатели кольцевого буфера так, что буфер представляется системе очищенным;
- записывает флаг Ctrl+Break в ячейку области данных BIOS по адресу 40h:71h;
- записывает код Ctrl+C (03h) в буфер драйвера консоли CON, что для драйвера консоли равносильно наличию кода Ctrl+C в кольцевом буфере ввода с клавиатуры.

В результате если после нажатия Ctrl+Break программа вызовет какую-либо функцию DOS, выполняющую проверку на Ctrl+C, то DOS обнаружит наличие Ctrl+Break и передаст управление на вектор 23h точно так же, как и при обнаружении

Ctrl+C. Таким образом, системная обработка Ctrl+Break и Ctrl+C в итоге выполняется почти одинаково. Особенность обработки Ctrl+C заключается в том, что если перед вводом Ctrl+C были нажаты какие-то клавиши и их коды остались в кольцевом буфере клавиатуры, они будут "маскировать" код Ctrl+C, так как драйвер консоли всегда анализирует только самый старый из символов, находящихся в кольцевом буфере. С вводом Ctrl+Break ситуация иная. Программа DOS, обрабатывающая прерывание 1Bh, отправляет код 03h не в кольцевой буфер клавиатуры, а непосредственно в драйвер CON. Поэтому комбинацию Ctrl+Break нельзя замаскировать вводом символов с клавиатуры (к тому же программа обработки Ctrl+Break очищает кольцевой буфер).

Прикладная программа может заменить содержимое вектора 1Bh адресом собственного обработчика. В этом случае при нажатии Ctrl+Break произойдет немедленный (через системный обработчик прерывания 09h и вектор 1Bh) переход на программу обработчика, который, таким образом, может взять на себя управление практически в любой точке программы, в том числе и при зацикливании или других чисто процессорных операциях. Однако такой обработчик работает на уровне прерываний, что ограничивает его возможности. В момент прерывания могли исполняться какие-то программы DOS, поэтому завершение обработчика иначе, чем командой iret, может привести к аварии системы. Кроме того, из обработчика нельзя обращаться к функциям DOS. Однако при всех этих ограничениях возможность в любой момент вмешаться в ход выполнения программы оказывается для некоторых приложений весьма полезной.

Исходное содержимое вектора 1Bh (в отличие от вектора 23h) не восстанавливается системой автоматически при завершении программы. Поэтому в прикладной программе, перехватывающей прерывание по Ctrl+Break, необходимо предусмотреть перед ее завершением восстановление исходного содержимого этого вектора. Однако пользователь может завершить программу и аварийно, нажав Ctrl+C и обойдя тем самым строки нормального завершения. Поэтому в программе, перехватывающей вектор 1Bh, следует предусмотреть собственный обработчик прерывания по Ctrl+C, в котором перед завершением программы восстанавливается вектор 1Bh.

Вернемся к программе 40.1, которая позволяет поставить весьма наглядный эксперимент, демонстрирующий системные алгоритмы обработки Ctrl+C и Ctrl+Break. Запустив программу, нажмем на любую клавишу. Это действие занесет код нажатой клавиши в кольцевой буфер ввода. Если после этого ввести команду Ctrl+C, аварийного завершения программы не произойдет, так как предварительно введенный код будет маскировать код 03 и функция DOS (в данном случае функция 02h) не будет его видеть. Если, однако, после этого ввести команду Ctrl+Break, программа немедленно завершится. Таким образом, команда Ctrl+Break является, можно сказать, более надежной, причем для ее реализации нет необходимости вводить в программу прикладной обработчик прерывания 1Bh – команда Ctrl+Break сама передаст управление через вектор 23h, активизировав наш обработчик этого прерывания.

Статья 41. **Резидентные программы**

Многие программы, обеспечивающие функционирование вычислительной системы (драйверы устройств, программы сжатия или шифрования данных, русификаторы, интерактивные справочники и др.), должны постоянно находиться в памяти и быстро реагировать на запросы пользователя или на какие-то события, происходящие в вы-

числительной системе. Такие программы носят название программ, резидентных в памяти (Terminate and Stay Resident, TSR), или просто резидентных. Сделать резидентной можно любую программу, однако ввиду того, что резидентная программа должна быть максимально компактной, чаще всего в качестве резидентных используют программы типа .COM.

Структура типичной резидентной программы выглядит следующим образом:

```
text    segment
        assume CS:text,DS:text
        org   100h
main    proc
        jmp   init          ;Переход на секцию инициализации
        ;Данные резидентной секции программы
        ...
entry:  ;Текст резидентной секции программы
        ...
main    endp
init    proc                ;Секция инициализации
        ...
        mov   DX,(init-main+10Fh)/16;Размер в параграфах
        mov   AH,3100h      ;Функция "Завершить и оставить в
        int   21h           ;памяти"
init    endp
text    ends
        end   main
```

Программа пишется в формате .COM, поэтому в ней предусматривается только один сегмент, с которым связываются сегментные регистры CS и DS; в начале сегмента резервируется 100h байт для PSP.

При запуске программы с клавиатуры управление передается (в соответствии с параметром директивы end) на начало процедуры main. Командой jmp сразу же осуществляется переход на секцию инициализации, которая может быть оформлена в виде отдельной процедуры или входить в состав процедуры main. В секции инициализации выполняются некоторые начальные действия, о которых мы поговорим позже. Последними строками секции инициализации вызывается функция DOS 31h, которая выполняет завершение программы с оставлением в памяти указанной ее части. Размер резидентной части программы (в параграфах) передается DOS в регистре DX. Определить этот размер можно, например, следующим образом. К разности смещений init-main, которая равна длине резидентной части программы в байтах, прибавляется размер PSP (100h) и еще число 15 (Fh), чтобы после целочисленного деления на 16 (для получения размера программы в параграфах) результат был округлен в большую сторону.

С целью экономии памяти секция инициализации располагается в конце программы и отбрасывается при ее завершении.

Функция 31h, закрепив за резидентной программой необходимую для ее функционирования память, передает управление командному процессору (как и обычная функция завершения программы 4Ch), и вычислительная система переходит в исходное состояние. Наличие программы, резидентной в памяти, никак не отражается на ходе вычислительного процесса, за исключением того, что уменьшается объем свободной памяти. Одновременно в память может быть загружено любое число резидентных программ. Процесс первичного запуска резидентной программы, приводящего к ее загрузке в память, обычно называют установкой программы.

На рис. 41.1 показаны элементы резидентной программы и их взаимодействие.

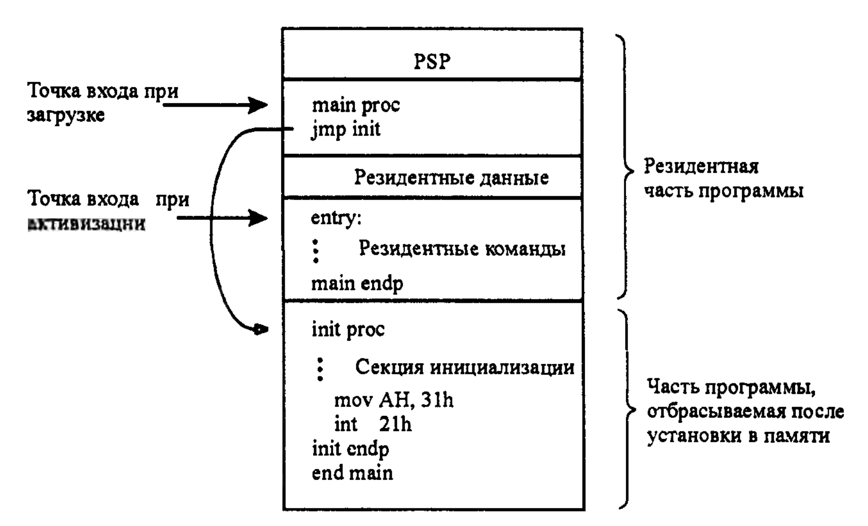


Рис. 41.1. Взаимодействие элементов резидентной программы

Любая резидентная программа имеет по крайней мере две точки входа. При запуске с клавиатуры программы типа .COM управление всегда передается на первый байт после PSP (IP=100h). Поэтому практически всегда первой командой резидентной программы является команда jmp, передающая управление на начало секции инициализации.

После отработки функции DOS 31h программа остается в памяти в пассивном состоянии. Для того чтобы активизировать резидентную программу, ей надо как-то передать управление. Вызвать к жизни резидентную программу можно разными способами, но наиболее употребительным является механизм аппаратных или программных прерываний. В этом случае в процессе инициализации необходимо заполнить соответствующий вектор адресом точки входа в программу (entry на рис. 41.1). Адрес entry образует вторую точку входа в программу, через которую осуществляется ее активизация. Очевидно, что резидентная секция программы должна заканчиваться командой выхода из прерывания iret.

В статье 37 был рассмотрен инструментальный обработчик программных прерываний, который активизировался командой int 3 и выводил на экран содержимое регистра флагов в точке своего вызова. Для использования этого обработчика его приходилось встраивать в отлаживаемую программу, что, конечно, очень неудобно. Преобразуем пример из статьи 37, сделав обработчик (вместе со всеми относящимися к нему подпрограммами) резидентным (пример 41.1). Это позволит нам "развязать" отлаживаемую программу с разработанным инструментальным средством, существенно повысив ценность последнего.

Пример 41.1. Резидентный обработчик прерывания 03h

```
.386
text    segment use16
        assume CS:text,DS:text
        org   100h
main    proc
        jmp   init
main    endp
flags   db 'FLAGS=****h $'
new_03h proc
```

```
        ...
new_03h endp
wrd_asc proc
        ...
wrd_asc endp
bin_asc proc
        ...
bin_asc endp
init    proc
        mov   AX,2503h
        mov   DX,offset new_03h
        int   21h
        mov   AH,09h
        mov   DX,offset msg
        int   21h
        mov   AH,31h
        mov   DX,(init-main+10Fh)/16
        int   21h
init    endp
msg     db 'Обработчик прерывания 3 загружен',13,10,'$'
text    ends
        end   main
```

Практически все элементы программы уже были рассмотрены; отметим только отличия от примера 37.1. Поле данных flags, которое должно быть резидентным, переместились в начало программы после команды jmp, чтобы пример точнее соответствовал рис. 41.1. С таким же успехом можно было оставить это поле на прежнем месте. При заполнении в секции инициализации вектора 03h не возникает проблем с перенастройкой регистра DS, так как в программе типа .COM все регистры указывают на единственный сегмент программы. Помимо установки вектора, в секции инициализации предусмотрен, как это обычно делается, вывод на экран сообщения о загрузке программы в память.

После запуска программы она остается в памяти и ничего не делает. Если, однако, в какой-либо из запущенных пользователем программ встретится команда int 3, наша резидентная программа активизируется и выведет на экран текущее для вызывающей программы содержимое регистра флагов. Таким образом, для испытания резидентного обработчика прерывания следует после установки его в памяти запустить тестовую программу с одной или несколькими командами int 3. Содержательная часть возможного варианта такой тестовой программы приведена в примере 41.2.

Пример 41.2. "Отлаживаемая программа"

```
main    proc
        int   3           ;FLAGS=3202h
        stc               ;Установим флаг CF
        int   3           ;FLAGS=3203h
        mov   AH,01h      ;Остановим программу
        int   21h         ;для наблюдения результата
        mov   AX,4C00h    ;Завершим тестовую программу
        int   21h
main    endp
```

Статья 42. **Защита резидентных программ от повторной установки**

Рассмотренная выше резидентная программа имеет серьезный недостаток: она не защищена от повторной загрузки. Вспомним, что для работы с резидентной программой ее следует прежде всего установить, запустив с клавиатуры как обычную программу. В этом случае управление передается на секцию инициализации, в которой подготавливаются условия для дальнейшей работы программы в резидентном состоянии. Как правило, в секции инициализации загружаются векторы прерываний, через которые программа будет активизироваться.

Может получиться, что пользователь, установив программу, забудет об этом и запустит ту же программу повторно. В этом случае в память будет загружена и останется резидентной вторая копия той же программы. Это плохо не только потому, что понапрасну расходуется память; более неприятным является вторичный перехват тех же векторов. Если резидентная программа после ее активизации не обращается к старому содержимому перехваченных ею векторов, то вторая копия полностью лишит первую работоспособности и тогда повторная загрузка приведет только к расходованию памяти. Если, однако, как это обычно и имеет место, резидентная программа в процессе своей работы передает управление старому обработчику перехваченного ею прерывания, то новая копия резидентной программы, сохранившая в процессе инициализации адрес первой копии в качестве содержимого перехватываемого вектора, будет при каждой активизации вызывать и первую копию. В результате резидентная программа будет фактически выполняться при каждом вызове дважды. Во многих случаях такое повторное выполнение нарушит правильную работу программы. Поэтому обязательным элементом любой резидентной программы является процедура защиты ее от повторной загрузки (установки).

Наиболее распространенным методом защиты резидентной программы от повторной установки является использование прерывания 2Fh, называемого мультиплексным и специально предназначенного для связи с резидентными программами. При вызове этого прерывания в регистре AH задается номер функции (от 00h до FFh), а в регистре AL – номер подфункции (в том же диапазоне). Функции с 00h по BFh зарезервированы для использования системой (например, функции 00h и 01h закреплены за резидентной программой DOS PRINT.COM, а 10h – за программой SHARE.EXE), а функции с C0h по FFh могут использоваться прикладными программами.

Для того чтобы резидентная программа могла отозваться на вызов прерывания int 2Fh, в ней должен иметься обработчик этого прерывания (рис. 42.1).

Фактически все резидентные программы, как системные, так и прикладные, имеют такие обработчики, через которые осуществляется не только проверка на повторную установку, но и вообще вся связь с резидентной программой: смена режима ее работы или получение от нее требуемой информации. Задание действия, которое надлежит выполнить обработчику прерывания 2Fh конкретной резидентной программы, осуществляется с помощью номера подфункции, помещаемого перед вызовом прерывания в регистр AL. Обычно для проверки на наличие в памяти используется подфункция 00h.

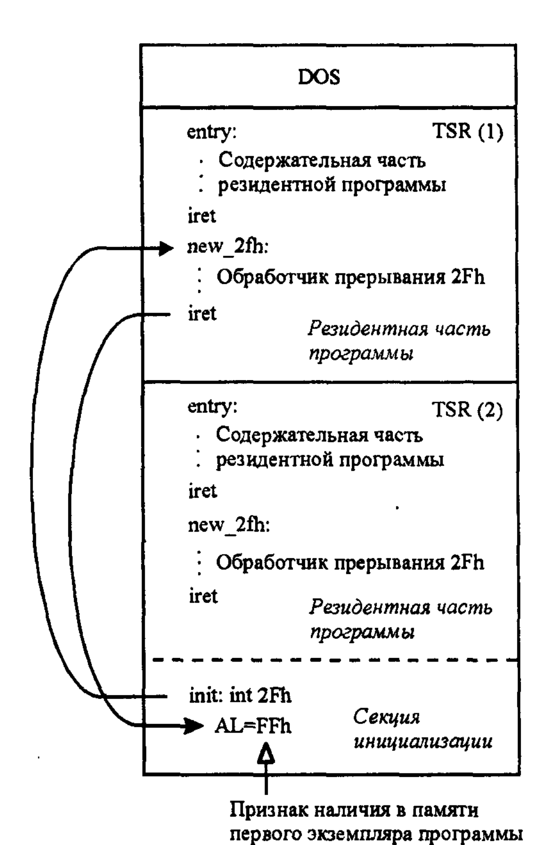


Рис. 42.1. Проверка, осуществляемая запущенной с клавиатуры резидентной программой, на наличие в памяти ранее установленного экземпляра той же программы

Резидентная программа на этапе инициализации должна выяснить, нет ли уже в памяти ранее установленного экземпляра той же программы. Для этого в секции инициализации выполняется команда int 2Fh (см. рис. 42.1). Предварительно в регистр AH помещается номер функции, присвоенный данной программе (из диапазона C0h...FFh), а в регистр AL – номер подфункции, соответствующей запросу на наличие в памяти, например, 00h.

Обработчик прерывания 2Fh резидентной программы должен прежде всего проверить номер функции в регистре AH; при обнаружении своей функции обработчик анализирует содержимое регистра AL и выполняет затребованные действия, после чего командой iret передает управление вызвавшей его программе. Если, однако, обработчик обнаружил в регистре AH чужую функцию, он должен командой jmp CS:old_2fh передать управление по цепочке тому обработчику, адрес которого был ранее в векторе 2Fh. В результате вызов int 2Fh из любой программы будет проходить по цепочке через все загруженные резидентные программы, пока не достигнет "своей" программы или не вернет управление в вызвавшую программу через обработчик DOS (который, очевидно, всегда будет самым последним в цепочке).

Естественно, для коммуникации с резидентной программой должен быть определен некоторый интерфейс. Обычно при проверке на повторную установку резидентная программа, если она уже находится в памяти, возвращает в регистре AL значение FFh, которое является признаком запрета вторичной загрузки. Именно эта ситуация изображена на рис. 42.1.

Иногда для большей надежности идентификации своей функции резидентная программа, помимо значения FFh в регистре AL, возвращает еще какие-то обусловленные заранее коды в других регистрах. Часто через дополнительные регистры передается символьная информация, например имя программы. В этом случае, если вызвавшая программа с именем, например, DUMP.COM получает после вызова int 2Fh в регистре AL значение FFh, а в регистрах CX и DX – символьные коды 'DU' и 'MP', она может быть уверена, что ее первая копия уже находится в памяти. Если же в регистре AL вернулся код FFh, а в регистрах CX и DX – коды, например, 'OK' и 'RB', это скорее всего означает, что закрепленная за нашей программой функция мультиплексного прерывания уже используется другой резидентной программой. В этом случае стоит сменить функцию, чтобы не возбуждать конфликтных ситуаций.

Устанавливаемая резидентная программа, выполнив команду мультиплексного прерывания int 2fh, должна проанализировать содержимое регистра AL. Если AL=FFh, т. е. сделана попытка повторной установки программы, секция инициализации завершает выполнение программы вызовом функции DOS 4Ch и повторной установки не происходит. Если же в AL вернулось исходное значение 0, это говорит о том, что вызов int 2fh не был перехвачен первой копией устанавливаемой программы, которой, следовательно, не существует, и секция инициализации приступает к процедуре установки программы – заполнению векторов (в том числе и вектора 2Fh), выводу на экран информационного сообщения и, наконец, завершению программы функцией DOS 31h – завершить и оставить в памяти.

Разумеется, защитить от повторной установки можно (и нужно!) любую резидентную программу. Рассмотрим эту методику на относительно простом примере резидентного обработчика аппаратных прерываний от системного таймера (пример 42.1). Такой обработчик можно использовать, например, в автоматизированной установке для периодического вывода на экран некоторой информации: времени, оставшегося до конца текущего сеанса измерений, показаний измерительных приборов, включенных в установку, и т. д. Мы для простоты ограничимся выводом на экран мерцающего символа, который будет свидетельствовать о наличии в памяти и нормальном функционировании резидентной программы.

Пример 42.1. Защита резидентной программы от повторной установки

```
text    segment
        assume CS:text,ds:text
org     256                ;Место для PSP
begin:  jmp    init        ;Переход на инициализацию
old_08h dd     0           ;Ячейки для исходного содержимого
old_2fh dd     0           ;перехватываемых векторов
sym1    dw     421Eh       ;Выводимый символ
sym2    dw     241Eh       ;То же с инверсным атрибутом
count   db     0           ;Ячейка для пересчета прерываний
;Обработчик прерываний от таймера
new_08h proc
        pushf              ;Флаги в стек
        call   CS:old_08h  ;Переход в системный обработчик с возвратом
```

```
        push    AX            ;Сохраним регистры AX и ES
        push    ES            ;
        inc     CS:count      ;Инкремент счетчика прерываний
        test    byte ptr CS:count,07h;Пересчет на 8
        jnz     exit          ;Если не 8-е, на выход
        mov     AX,0B800h     ;Настроим ES
        mov     ES,AX         ;на сегмент видеопамяти
        mov     AX,CS:sym1    ;Получим символ с атрибутом из sym1
        mov     ES:3998,AX    ;Выведем в последнюю позицию экрана
        xchg    AX,CS:sym2    ;Обменяем содержимое ячеек
        mov     CS:sym1,AX    ;sym1 и sym2
exit:   mov     AL,20h        ;Стандартное завершение аппаратного
        out     20h,AL        ;обработчика прерываний - EOI
        pop     ES            ;и восстановление
        pop     AX            ;сохраненных регистров
        iret                  ;Выход из прерывания
new_08h endp
;Обработчик мультиплексного прерывания
new_2fh proc
        cmp     AX,0C800h     ;Наша функция с подфункцией 00h?
        jne     out_2fh       ;Не наша функция или подфункция
        mov     AL,0FFh       ;Все наше, сообщим "Я уже в памяти"
        iret                  ;Возврат в вызвавшую программу
out_2fh:jmp     CS:old_2fh    ;Вызов следующего обработчика
new_2fh endp
;Конец резидентной части программы и начало секции инициализации
init    proc
;Проверка на наличие в памяти 1-го экземпляра этой же программы
        mov     AX,0C800h     ;Функция C8h, подфункция 00h
        int     2Fh           ;Вызов обработчика прерывания 2Fh
        cmp     AL,0FFh       ;Вернулся код FFh?
        jne     ok            ;Нет, можно устанавливать программу
        mov     AH,09h        ;Да, такая программа уже есть
        mov     DX,offset msg2;Выведем сообщение
        int     21h
        mov     AX,4C00h      ;и завершим программу обычным образом,
        int     21h           ;без оставления в памяти
;1-го экземпляра в памяти не оказалось. Установим программу
ok:     mov     AX,3508h      ;Чтение и сохранение вектора 8
        int     21h
        mov     word ptr CS:old_08h,BX
        mov     word ptr CS:old_08h+2,ES
        mov     AX,352Fh      ;Чтение и сохранение вектора 2Fh
        int     21h
        mov     word ptr CS:old_2fh,BX
        mov     word ptr CS:old_2fh+2,ES
        mov     AX,2508h      ;Установка обработчика 8
        mov     DX,offset new_08h
        int     21h
        mov     AX,252Fh      ;Установка обработчика 2Fh
        mov     DX,offset new_2fh
        int     21h
        mov     AH,09h        ;Выведем сообщение об успешной
        mov     DX,offset msg1;установке программы
        int     21h
        mov     AX,3100h      ;Завершим и оставим в памяти
        mov     DX,(init-begin+10Fh)/16
        int     21h
init    endp
;Поля данных в нерезидентной части программы
msg1    db      'Резидентный обработчик установлен$'
```

```
msg2    db    'Попытка вторичной установки. Установка отменена$'
text    ends
        end   begin
```

Резидентная часть программы включает два обработчика прерываний. Обработчик new_08h аппаратного прерывания от таймера является "содержательным", он реализует назначение программы – периодический вывод на экран мерцающего символа. Обработчик new_2fh программного мультиплексного прерывания является инструментальным – он служит лишь для обеспечения проверки программы на ее наличие в памяти.

Среди свободных функций мультиплексного прерывания мы произвольно выбрали для нашей программы функцию C8h, а для проверки на повторную установку использовали подфункцию 00h. Резидентный обработчик прерывания 2Fh, включенный в нашу программу, проверяет номера функции и подфункции и при обнаружении каких-либо других кодов передает управление следующему обработчику этого прерывания. Если же вызвана функция C8h с подфункцией 00h, обработчик устанавливает в регистре AL значение FFh ("Я уже загружен") и возвращает управление в вызвавшую программу командой iret.

Секция инициализации начинается с проверки на наличие в памяти. После загрузки в регистр AH номера функции (C8h), а в регистр AL – номера подфункции (00h) вызывается прерывание 2Fh. После возврата из прерывания анализируется содержимое регистра AL. Если обработчик вернул значение FFh, программа выводит предупреждающее сообщение и завершается без оставления в памяти обычной функцией завершения 4Ch. Если возвращено другое значение, происходит переход на метку ok и выполняются действия по установке программы в памяти (для надежности стоило проверить, возвращен ли именно 0). Сохраняется старое содержимое векторов 08h и 2Fh, устанавливаются наши обработчики этих прерываний, затем выводится сообщение об успешной загрузке, и программа завершается вызовом функции DOS 31h.

Включенный в программу резидентный обработчик прерываний от таймера должен периодически выводить в заданное место экрана мерцающий символ. Процедура обработчика new_08h начинается с вызова системного обработчика, чем осуществляется сцепление прикладного обработчика с системным и обеспечение их совместного функционирования (сначала выполняется программа системного обработчика, затем – прикладного). Далее выполняется алгоритм снижения в 8 раз частоты мерцания символа. Для отсчета прерываний предусмотрена резидентная ячейка count. При каждой активизации обработчика (18 раз в секунду) содержимое этой ячейки командой inc увеличивается на единицу. Следующая далее пара команд

```
test  byte ptr CS:count,07h
jnz exit
```

осуществляет переход на метку exit, т. е. на завершение обработки данного прерывания, если установлен хотя бы один из битов 0, 1 или 2 байта count. Таким образом, дальнейшие предложения программы обработчика будут выполняться лишь в тех случаях, когда все эти 3 бита сброшены. При последовательном наращивании счетчика count такая ситуация будут возникать при каждом восьмом вызове обработчика.

Далее в регистр ES заносится сегментный адрес начала видеопамяти (0B800h) и на последнее знакоместо экрана со смещением 3998 выводится содержимое ячейки sym1 – символ вместе с его атрибутом. Вывод одного и того же символа в одно и то же место экрана приведет к тому, что мы не будем знать, работает ли наш обработчик. В примере предусмотрена периодическая смена атрибута символа, что делает символ

мерцающим. Для этого среди резидентных данных предусмотрены две ячейки sym1 и sym2 с одинаковыми кодами символа, но разными атрибутами. При каждом проходе программы обработчика эти ячейки обмениваются своим содержимым. В результате на экран выводится то один код, то другой. Для обмена содержимого регистров или ячеек памяти в составе команд микропроцессора имеется команда xchg (exchange, обмен). Поскольку команды с адресацией обоих операндов в памяти запрещены, обмен приходится осуществлять через регистр AX.

По метке exit осуществляется стандартный выход из аппаратного прерывания. В контроллер прерываний посылается команда EOI, восстанавливаются сохраненные ранее регистры и командой iret осуществляется возврат в прерванную программу.

Рассмотренный пример будет правильно работать в чистой DOS, а также в сеансах командной строки систем Windows 3.1, NT и 2000. В системах Windows 95/98 прерывание 2Fh перехватывается системой и передается только тем резидентным программам, которые были загружены до загрузки самой Windows, в глобальном экземпляре DOS. Те же резидентные программы, которые загружаются в локальных сеансах DOS из системы Windows, пользоваться интерфейсом 2Fh не могут.

Статья 43. **Выгрузка резидентных программ из памяти**

Еще один серьезный недостаток, присущий рассмотренным ранее резидентным программам, заключается в невозможности выгрузить их из памяти, если отпала необходимость в их использовании. Как уже отмечалось, в DOS нет средств выгрузки резидентных программ. Единственный предусмотренный для этого механизм – перезагрузка компьютера. Практически, однако, большинство резидентных программных продуктов имеют встроенные средства выгрузки. Активизация этих средств осуществляется с помощью уже рассмотренного выше мультиплексного прерывания 2Fh. Если встроенный в резидентную программу обработчик этого прерывания, анализируя номер подфункции (содержимое регистра AL), обнаруживает, что этот номер соответствует команде выгрузки, он реализует все действия, необходимые для выгрузки программы из памяти.

Вызов прерывания 2Fh, реализующий выгрузку конкретной резидентной программы, можно выполнить в специально созданной выгружающей программе. Практически всегда в качестве выгружающей используют саму резидентную программу, точнее, ее вторую копию, которая, если ее запустить в определенным режиме, не только не пытается остаться в памяти, но, наоборот, выгружает из памяти свою первую копию.

Собственно выгрузка резидентной программы осуществляется очень просто. Достаточно освободить блоки памяти, занимаемые программой (собственно программой и ее окружением, см. статью 14) с помощью функции DOS 49h. Однако перед освобождением памяти необходимо восстановить все векторы прерываний, перехваченные резидентной программой. Следует подчеркнуть, что восстановление векторов представляет в общем случае значительную и иногда даже неразрешимую проблему. Во-первых, старое содержимое вектора, которое хранится где-то в полях данных резидентной программы, невозможно извлечь снаружи, из другой программы, так как нет никаких способов определить, где именно его спрятала резидентная программа в процессе инициализации. Поэтому выгрузку резидентной программы легче осуществить

из нее самой, чем из другой программы. Во-вторых, даже если выгрузку осуществляет сама резидентная программа, она может правильно восстановить старое содержимое вектора лишь в том случае, если этот вектор не был позже перехвачен другой резидентной программой. Если же это произошло, в таблице векторов находится уже адрес не выгружаемой, а следующей резидентной программы, и, если восстановить старое содержимое вектора, эта следующая программа "зависнет", лишившись средств своего запуска. Поэтому надежно можно выгрузить только последнюю из загруженных резидентных программ.

Модифицируем рассмотренную в предыдущей статье программу, чтобы ее можно было выгружать из памяти по команде с клавиатуры. Для этого, как и для защиты от повторной установки, используется мультиплексное прерывание 2Fh. Примем, что подфункция 00h прерывания 2Fh служит для проверки на наличие в памяти, а подфункция 01h – для выгрузки.

Изменения коснутся двух блоков программы: резидентного обработчика прерывания 2Fh и секции инициализации. Ниже приведены новые варианты этих блоков (пример 43.1).

Пример 43.1. Выгрузка резидентной программы командой с клавиатуры

```
;Резидентный обработчик мультиплексного прерывания
new_2fh proc
        cmp    AH,0C8h       ;Наша функция?
        jne    out_2fh       ;Не наша, назад по цепочке
        cmp    AL,00h        ;Наша; пришла команда 00h?
        je     iamhere       ;Да, сообщим "Я уже в памяти"
        cmp    AL,01h        ;Пришла команда 01h?
        je     uninst        ;Да, на блок выгрузки
out_2fh:jmp    CS:old_2fh    ;Пришла неизвестная команда
iamhere:mov    AL,0FFh       ;Сообщим о наличии
        iret                 ;и выход из прерывания
;Приступим к выгрузке программы из памяти
uninst: push   DS            ;Сохраним
        push   ES            ;используемые
        push   DX            ;регистры
        mov    AX,2508h      ;Восстановим вектор 8
        lds    DX,CS:old_08h;из ячейки, где он был
        int    21h           ;сохранен при инициализации
        mov    AX,252Fh      ;Восстановим вектор 2Fh
        lds    DX,CS:old_2fh;из ячейки, где он был
        int    21h           ;сохранен при инициализации
        mov    ES,CS:[2Ch]   ;Сегмент окружения из PSP
        mov    AH,49h        ;Функция освобождения блока памяти
        int    21h
        push   CS            ;Скопируем CS в ES
        pop    ES            ;ES указывает на начало программы
        mov    AH,49h        ;Функция освобождения блока памяти
        int    21h
        pop    DX            ;Восстановим
        pop    ES            ;сохраненные ранее
        pop    DS            ;регистры
        iret                 ;Возврат в вызывающую программу
new_2fh endp
;Секция инициализации
init    proc
        mov    AX,0C800h     ;Проверка на наличие в памяти
        int    2Fh           ;1-го экземпляра программы
        cmp    AL,0FFh       ;Вернулся код FFH?
```

```
        jne    ok            ;Нет, этот экземпляр 1-й.
;1-й экземпляр обнаружен. Был ли у команды параметр 'off'?
        mov    CL,ES:80h     ;Получим длину хвоста команды из PSP
        cmp    CL,0          ;Длина хвоста=0?
        je     fin           ;Да, команда без параметра
        xor    CH,CH         ;Команда с параметром. CX=длина хвоста
        mov    DI,81h        ;ES:DI→хвост в PSP
        mov    SI,offset tail;DS:SI→поле tail
        mov    AL,' '        ;Уберем пробелы из начала хвоста
repe    scasb                ;Сканируем хвост, пока пробелы
        dec    DI            ;DI→первый символ после пробелов
        mov    CX,3          ;Ожидаемая длина параметра
repe    cmpsb                ;Сравниваем введенный параметр с ожидаемым
        jne    fin           ;Введена ошибочная команда, на выход
        mov    AX,0C801h     ;Пошлем в резидентную программу команду
        int    2Fh           ;на выгрузку 01h
        mov    DX,offset msg3;Выведем сообщение об этом
        jmp    fin1
fin:    mov    DX,offset msg2 ;Выведем сообщение msg2
fin1:   mov    AH,09h
        int    21h
        mov    AX,4C00h      ;Завершим программу без
        int    21h           ;оставления в памяти
;1-го экземпляра в памяти не оказалось. Установим программу
ok:
        ...                  ;Далее по тексту примера 42.1
init    endp
;Поля данных в нерезидентной части программы
msg1    db     'Резидентный обработчик установлен$'
msg2    db     'Попытка вторичной установки. Установка отменена$'
msg3    db     'Программа успешно выгружена из памяти$'
tail    db 'off'             ;Ожидаемый параметр команды
```

Резидентный обработчик прерывания 2Fh прежде всего проверяет номер функции, поступивший в регистре AH. Если этот номер отличается от C8h, управление передается следующему обработчику по цепочке. Далее анализируется содержимое регистра AL. Если AL=00h, осуществляется переход на метку iamhere, где в регистр AL засылается код FFh наличия в памяти первого экземпляра данной программы, после чего командой iret управление возвращается в вызывающую программу. Если AL=01h, осуществляется переход на метку uninst для выполнения действий по выгрузке программы. При любом другом номере подфункции управление передается следующему обработчику по цепочке.

По метке uninst осуществляется сохранение используемых далее регистров (что делается скорее для красоты, чем по необходимости), и функцией DOS 25h восстанавливается из ячеек old_08h и old_2Fh исходное содержимое соответствующих векторов. Далее из ячейки со смещением 2Ch относительно начала PSP в ES загружается адрес окружения программы. Сегментный адрес освобождаемого блока памяти – единственный параметр, требуемый для выполнения функции DOS 49h (размер освобождаемого блока DOS известен, он хранится в блоке управления памятью MCB). Далее освобождается блок памяти с самой программой. Сегментный адрес этого блока (адрес PSP) находится, естественно, в CS. Наконец, командой iret управление передается в программу, вызвавшую прерывание 2Fh.

Функция 49h оповещает DOS о том, что данный блок памяти свободен и может впредь использоваться DOS. Это, однако, не мешает выполняться завершающим стро-

кам программы (в данном случае – команде iret), поскольку логическое освобождение памяти, выполняемое функцией 49h, не разрушает ее содержимого. Наша резидентная программа физически сотрется лишь после того, как в память будет загружена очередная выполняемая программа.

Перед тем как перейти к рассмотрению секции модифицированной секции инициализации, поясним смысл использованного в ней алгоритма. Вообще говоря, для того, чтобы удалить из памяти резидентную программу, достаточно в какой-то программе вызвать прерывание 2Fh с функцией, присвоенной нашей программе, и подфункцией выгрузки 01h. Проще всего создать для этого специальную выгружающую программу (пример 43.2).

Пример 43.2. Программа выгрузки резидентной программы

```
text    segment
        assume CS:text
begin:  mov    AX,0C801h
        int    2Fh
        mov    AX,4C00h
        int    21h
text    ends
        end    begin
```

Любопытно, что в выгружающей программе не указывается имя выгружаемой. Выгружается та резидентная программа, которой при ее разработке была назначена функция C8h мультиплексного прерывания.

Очевидно, что методика выгрузки резидентной программы с помощью специально предназначенной для этого выгружающей программы довольно неуклюжа. Для каждой резидентной программы нам придется создавать выгружающую. Гораздо изящнее использовать в качестве выгружающей саму резидентную программу. Обычно в ней предусматривают анализ запускающей программу командной строки так, чтобы ввод с клавиатуры имени программы загружал эту программу в память, оставляя ее резидентной, а ввод имени программы с параметром off приводил к выгрузке программы из памяти. Именно такой механизм реализован в примере 43.1.

Если программа запускается с клавиатуры с указанием каких-либо параметров (имен файлов, ключей, определяющих режим работы программы и пр.), то DOS, загрузив программу в память, помещает все символы, введенные после имени программы, (так называемый хвост команды), в префикс программного сегмента программы начиная с относительного адреса 80h. Хвост команды помещается в PSP во вполне определенном формате. В байт по адресу 80h DOS заносит число символов в хвосте команды (включая пробел, разделяющий на командной строке саму команду и ее хвост). Далее (начиная с байта по адресу 81h) следуют все символы, введенные с клавиатуры до нажатия клавиши Enter. Завершается хвост кодом возврата каретки (13).

Таким образом, если программа с именем, например, 43-01 была вызвана командой

```
43-01 off
```

то в PSP, начиная с байта 80h, будет записана следующая информация:

```
4,' off',13
```

Вернемся к примеру 43.1 и рассмотрим алгоритм анализа хвоста команды, который реализован в секции инициализации.

Прежде всего с помощью функции C800h прерывания 2Fh выполняется проверка на наличие в памяти первого экземпляра резидентной программы. Если первый экзем-

пляр не обнаружен, то независимо от вида запускающей программу команды (с параметром или без него) происходит переход на метку ok и установка программы в памяти. В противном случае начинается анализ хвоста команды. При запуске программы типа .COM все сегментные регистры указывают на начало PSP. Байт с длиной хвоста (возможно, нулевой), находящийся по адресу ES:80h, помещается в регистр CL и сравнивается с нулем. Если в нем 0, команда запуска была введена без параметров. В этом случае, как и в предыдущем примере, выводится сообщение о невозможности вторичной установки и инициализация завершается вызовом функции 4Ch. Если же хвост имеет ненулевую длину, следует проанализировать его состав.

Обнулением регистра CH длина хвоста расширяется на весь регистр CX, что нужно для организации цикла. Регистр DI настраивается на первый байт хвоста, а регистр SI – на начало поля tail с ожидаемой формой параметра (строка 'off'). Регистр AL подготавливается для выполнения команды сканирования строки. Команда scasb сравнивает в цикле байты хвоста с содержимым AL (кодом пробела). Сравнение ведется до тех пор, пока не будет найден первый символ, отличный от пробела. Эта операция необходима из-за того, что оператор при вводе команды выгрузки может отделить параметр команды от самой команды любым числом пробелов, которые попадут в хвост команды в PSP и помешают анализу введенного параметра.

Выход из цикла выполнения команды scasb осуществляется, когда команда проанализировала первый после пробела символ. После этого регистр DI указывает на второй символ параметра. Команда dec DI корректирует указатель DI, направляя его на первый значащий символ введенного параметра. Далее командой сравнения строк cmpsb осуществляется сравнение трех оставшихся символов хвоста со строкой 'off'. Если результат сравнения оказался отрицательным (параметр есть, но он неправильный), осуществляется переход на завершение программы с предварительным выводом сообщения о невозможности повторной установки. Если же введенная команда содержала параметр 'off', в первый резидентный экземпляр программы посылается прерывание 2Fh с кодом 01h в регистре AL – команда на выгрузку. После того как первый экземпляр отработает эту команду (освободит от себя память), секция инициализации второго экземпляра завершается функций 4Ch с предварительным выводом сообщения об успешной выгрузке.

Составленная нами программа не лишена недостатков. Так, в ней анализируются всегда только три значащих символа хвоста. Таким образом, программа будет выгружена и при вводе, например, команды

```
43-01 offset
```

Другой недостаток заключается в том, что результат сравнения записанного в программе хвоста с введенным с клавиатуры параметром будет положительным, только если с клавиатуры введены строчные буквы. Команда

```
43-01 OFF
```

не приведет к выгрузке программы. По-настоящему следовало включить в программу перед анализом хвоста преобразование символов параметра в прописные буквы.

Статья 44. **Использование системных средств в обработчиках аппаратных прерываний**

При разработке обработчиков аппаратных прерываний, в особенности тех, которые входят в состав резидентных программ, возникает целый ряд проблем, которые, как правило, удается разрешить, однако часто ценой существенного усложнения программ. Некоторые из этих проблем относятся только к резидентным обработчикам или вообще к резидентным программам; другие свойственны как резидентным, так и транзитным обработчикам аппаратных прерываний.

Для обработчиков аппаратных прерываний характерен асинхронный способ активизации: аппаратное прерывание может возникнуть в любой момент времени и состояние программы на момент прерывания никогда не может быть известно заранее. В частности, обработчик аппаратного прерывания может получить управление в тот момент, когда в основной программе выполняется запрошенная ею функция DOS. Система DOS является нереентерабельной: нельзя, прервав выполнение какой-то функции DOS, вызвать ту же или другую функцию – это неминуемо приведет к разрушению системы. Поэтому без принятия специальных мер в обработчике аппаратного прерывания недопустимо обращаться к функциям DOS. Это исключает, например, работу с файлами, службой времени, клавиатурой. Вывод на экран нетрудно осуществить путем прямой записи в видеопамять, однако другие ресурсы компьютера оказываются практически недоступны.

Чтобы разобраться в причинах нереентерабельности DOS и в методах преодоления этого недостатка, мы должны рассмотреть некоторые вопросы внутренней организации DOS.

Среди системных областей DOS имеется весьма важная структура, которую мы будем называть областью текущих данных (в оригинальной литературе она носит название области выгружаемых данных (Swappable Data Area, SDA). Это довольно большая область объемом около 2 Кбайт, адрес которой можно получить с помощью функции DOS 5D06h:

```
mov   AX,5D06h
int   21h
```

Функция возвращает в регистрах DS:SI адрес SDA, а в регистре CX – ее размер. Наиболее интересные для прикладного программиста поля SDA представлены в табл. 44.1.

Таблица 44.1. Некоторые поля области текущих данных SDA

Смещение	*Число байтов*	*Назначение*
00h	1	Флаг критической ошибки (флаг ErrorMode)
01h	1	Флаг занятости DOS (флаг InDOS)
02h	1	Дисковод последней критической ошибки
04h	2	Код расширенной ошибки для последней ошибки
06h	1	Рекомендуемое действие для последней ошибки
08h	4	Содержимое ES:DI при последней ошибке
0Ch	2	Адрес текущей области дисковой передачи DTA
10h	2	Сегмент текущего PSP (ID текущей программы)

12h	2	Ячейка для хранения SP при вызове int 23h
14h	1	Код возврата последнего завершенного процесса
16h	1	Текущий дисковод
17h	1	Расширенный флаг BREAK
30h	1	День месяца
31h	2	Месяц
32h	2	Текущий год от 1980
34h	1	Номер дня от 01.01.1980
36h	1	День недели
264h	4	Прошлый кадр стека
268h	4	Указатель на текущую системную таблицу файлов SFT
284h	2	Относительный кластер в файле, к которому выполняется доступ
286h	2	Абсолютный номер кластера, к которому выполняется доступ
288h	2	Текущий номер сектора
28Ah	2	Текущий номер кластера
28Ch	2	Текущий относительный номер сектора в файле
336h	330	Вспомогательный стек (для функций 01h...0Ch при наличии критической ошибки)
480h	384	"Дисковый стек" (для функций 00h, 0Dh...6Ch)
600h	384	Стек ввода-вывода (для функций 01h...0Ch)

Рассмотрим поля области текущих данных, имеющие отношение к разработке обработчиков прерываний.

Флаг критической ошибки ErrorMode устанавливается DOS, если зафиксировано состояние критической ошибки, которое может возникнуть по разным причинам, в большинстве своем связанным с дисковыми операциями: попытка записи на защищенный или отсутствующий диск, на диске не найден требуемый сектор, ошибка в контрольной сумме данных в секторе и т. д. Обнаружив такую ситуацию, DOS устанавливает флаг ErrorMode, записывая в этот байт SDA 1 и выполняет команду int 24h. В этом векторе (содержимое которого дублируется в PSP каждой запускаемой задачи, см. табл. 32.1) хранится адрес системного обработчика критической ошибки, который выводит на экран аварийное сообщение и вводит ответную команду пользователя.

Флаг занятости DOS, обычно называемый флагом InDOS ("внутри DOS"), устанавливается диспетчером DOS сразу после анализа им номера вызванной функции DOS и сбрасывается перед возвратом из DOS в прикладную программу. Таким образом, значение этого флага, равное единице, говорит о том, что выполняется программа DOS. Функции DOS в прикладной программе можно вызывать, только если флаг InDOS сброшен.

Сегментный адрес текущего PSP (смещение 10h) является важнейшей системной переменной. В этом поле записывается адрес PSP (идентификатор ID) той программы, которую DOS считает текущей. Именно эта программа будет завершена, если выдать запрос на выполнение функции 4Ch (независимо от того, какая программа выдала этот запрос). Далее, при создании или открытии файлов используется таблица файлов задания JFT, находящаяся в PSP текущей программы, опять же независимо от того, какая программа реально выдала запрос на работу с файлом. Понятие текущей программы приобретает особую важность при разработке обработчиков аппаратных прерываний.

Действительно, при вызове обработчика изменяются только значения CS:IP, текущей же остается прерванная программа. Поэтому, например, дескрипторы файлов, открытых в обработчике, находятся не в PSP обработчика, а в PSP прерванной программы и при завершении этой программы работа обработчика (если он, будучи резидентным, остался в памяти) будет нарушена.

Адрес текущей области дисковой передачи – DTA (Disk Transfer Area) важен в тех случаях, когда в обработчике аппаратного прерывания вызываются функции DOS, использующие эту область. Раньше, когда для работы с файлами использовались блоки управления файлами (FCB), с помощью DTA осуществлялись все файловые операции; в настоящее время эта область нужна только при выполнении функций поиска файлов.

В области текущих данных находятся и все три системных стека – стек ввода-вывода, который используется DOS при выполнении функций ввода-вывода из диапазона 01h...0Ch, "дисковый" стек для всех остальных функций DOS (файловых, службы времени, управления памятью и процессами и др.) и вспомогательный стек, на который DOS переключается в том случае, если в процессе обработки критической ошибки возникла необходимость обратиться к функциям ввода-вывода (главным образом через терминал, т. е. к функциям ввода с клавиатуры и вывода на экран). Наличие внутренних стеков повышает надежность работы DOS, так как устраняется возможность переполнения стека прикладной программы при выполнении запросов к DOS, которые в своем большинстве активно используют стек для вызова системных подпрограмм, сохранения текущих значений и передачи параметров.

Диспетчер DOS, получив управление в результате выполнения команды int 21h, совершает целый ряд операций, из которых прежде всего надо отметить следующие:

- сохранение регистров задачи на стеке задачи;
- инкремент флага InDOS, который при первом (невложенном) вызове DOS становится равен единице;
- сохранение прошлого кадра стека (SS:SP) в ячейке для позапрошлого кадра;
- сохранение в ячейке для прошлого кадра стека SS:SP прерванной задачи;
- занесение в SS:SP кадра одного из системных стеков (в соответствии с вызванной функцией);
- вызов по номеру функции требуемой программы DOS.

После завершения вызванной функции диспетчер выполняет описанные шаги в обратном порядке: восстанавливает кадр стека задачи и кадр прошлого стека, декрементирует флаг InDOS, который опять становится равен нулю, восстанавливает из стека задачи ее регистры и выполняет команду iret возврата в прерванную задачу.

Нереентерабельность DOS связана с тем, что при выполнении большинства функций используется один и тот же системный стек. Если выполнение какой-либо функции DOS будет прервано аппаратным прерыванием и в обработчике прерывания будет вызвана функция DOS из той же группы, т. е. использующая тот же стек, то диспетчер загрузит в SS:SP в точности те же значения, что и при первом вызове. В результате вложенный вызов будет затирать те данные, которые были сохранены в стеке первым вызовом. Однако при вызове функции из другой группы ничего страшного не произойдет, так как функции ввода-вывода и "дисковые" работают на разных системных стеках. При этом наличие в SDA двух ячеек для хранения не только прошлого, ни и позапрошлого кадров стека позволяет правильно обрабатывать один вложенный вызов

DOS. Эта возможность широко используется во "всплывающих" программах, активизируемых с клавиатуры.

Местоположение флага занятости DOS можно получить с помощью функции DOS 34h, которая возвращает двухсловный адрес флага InDOS в регистрах ES:BX. Получив и сохранив этот адрес на этапе инициализации обработчика прерывания, в самом обработчике перед вызовом функций DOS следует выполнить проверку флага занятости. Будем считать, что адрес флага сохранен в ячейке in_dos:

```
in_dos   dd    0            ;Адрес флага InDOS
task_reqdb    0            ;Флаг требования запуска task
         ...
dos_busy:les   BX,CS:in_dos;Получили адрес флага InDOS
         cmp   ES:[BX],0   ;DOS свободна?
         jne   wait_dos    ;Нет, придется ждать
         call  task        ;Да, можно вызывать функции DOS
         iret              ;Завершим обработчик
wait_dos inc   CS:task_req ;Установим флаг требования запуска
         iret              ;Завершим обработчик
```

В этом фрагменте предполагается, что процедура task как раз и содержит вызовы DOS. Если флаг InDOS сброшен, вызывается эта процедура и после ее завершения завершается и весь обработчик. Если же флаг InDOS установлен, обработчик устанавливает флаг task_req, который говорит о том, что обработчик из-за занятости DOS "недовыполнил" свои функции и процедура task требует своего запуска. Управление возвращается в прерванную задачу (фактически – в прерванную функцию DOS), и DOS имеет возможность завершить свою работу.

Каким же образом можно теперь запустить процедуру task? Для этого нужен дополнительный "активизатор" нашего обработчика. В качестве такого активизатора естественно воспользоваться прерываниями от таймера. Таким образом, обработчик, помимо своей основной точки входа, должен иметь еще и точку входа для обработки прерываний от системного таймера:

```
in_dos   dd    0            ;Адрес флага InDOS
task_reqdb    0            ;Флаг требования запуска task
old_08h dd    0            ;Ячейка для хранения исходного содержимого вектора 08h
         ...
new_08h: pushf              ;Передадим управление в системный
         call  CS:old_08h  ;обработчик прерываний от таймера
         cmp   CS:task_req,1;Процедура task требует запуска?
         jne   out_08h     ;Нет, можно завершить обработку
         les   BX,CS:in_dos;Да, снова выясним,
         cmp   ES:[BX],0   ;свободна ли DOS?
         jne   out_08h     ;Нет, придется опять ждать
         dec   CS:task_req ;Да, сбросим флаг требования
         call  task        ;запуска и вызовем task
out_08h: iret              ;Завершим обработчик
```

Проверка занятости DOS по состоянию флага InDOS необходима, но недостаточна в тех ситуациях, когда возможно возникновение критической ошибки. Дело в том, что обнаружив критическую ошибку, DOS выполняет декремент флага InDOS и инкремент флага критической ошибки ErrorMode, находящегося, как и флаг InDOS, в области текущих данных. После этого DOS с помощью прерывания int 24h вызывает обработчик критической ошибки, который выводит на экран запрос, соответствующий сложившейся ситуации, например:

```
Not ready reading drive A
Abort, Retry, Fail?
```

и ждет указаний пользователя. И вывод на экран, и ввод с клавиатуры осуществляются с помощью функций DOS из диапазона 01h...0Ch, причем они вызываются "изнутри" той функции DOS, в процессе выполнения которой возникла критическая ошибка. Диспетчер DOS перед переключением стека проверяет состояние флага критической ошибки и, если этот флаг установлен, делает текущим не стек ввода-вывода, как обычно, а вспомогательный стек. В результате такой вложенный вызов DOS не приводит к разрушению системы.

Таким образом, в обработчике аппаратного прерывания перед вызовом какой-либо функции DOS следует анализировать состояние не только флага InDOS, но и флага ErrorMode. Если хотя бы один из них не равен нулю, вызывать DOS нельзя.

В версиях DOS отсутствуют документированные средства для определения адреса флага критической ошибки. Известно, однако, что в DOS начиная с версии 3.10 он расположен в первом байте области текущих данных перед флагом InDOS и для получения его адреса можно воспользоваться функцией 5D06h. Поскольку сама эта функция не является реентерабельной, определение адреса следует выполнить на этапе инициализации обработчика (вместе с определением адреса флага InDOS).

С учетом сказанного приведенный выше фрагмент проверки занятости DOS будет выглядеть следующим образом (предполагается, что адрес флага критической ошибки помещен в ячейку crit_err):

```
in_dos   dd    0              ;Адрес флага InDOS
crit_err dd    0              ;Адрес флага ErrorMode
task_req db    0              ;Флаг требования запуска task
         ...
dos_busy:les   BX,CS:in_dos;Получили адрес флага InDOS
         lds   SI,CS:crit_err;Получили адрес флага ErrorMode
         cmp   ES:[BX],0      ;Флаг InDOS сброшен?
         jne   wait_dos       ;Нет, придется ждать
         cmp   DS:[SI],0      ;Флаг ErrorMode сброшен?
         jne   wait_dos       ;Нет, придется ждать
         call  task           ;Да, оба флага сброшены,
                              ;можно вызывать функции DOS
         iret                 ;Завершим обработчик
wait_dos inc   CS:task_req    ;Установим флаг требования запуска
         iret                 ;Завершим обработчик
```

Аналогичную коррекцию следует ввести и в процедуру обработки прерываний от таймера.

Рассмотрим пример использования средств DOS в обработчике аппаратных прерываний (пример 44.1). Основная программа в бесконечном цикле выводит на экран некоторое сообщение функцией DOS 40h. Обработчик прерываний от клавиатуры анализирует скан-код нажатой клавиши и при поступлении кода клавиши "серый плюс" активизирует процедуру task, которая получает из системы текущее время и после преобразования в символьную форму выводит его на экран. Чтение времени осуществляется функцией DOS 2Ch, вывод на экран – функцией DOS 09h. Поскольку активизация обработчика с большой вероятностью выполняется в тот момент, когда основная программа выполняет функцию DOS 40h, вызов других функций DOS в таком обработчике недопустим. Для того чтобы получить возможность обращаться к DOS в обработчике прерываний от клавиатуры, в нем следует использовать описанный выше алгоритм анализа занятости DOS.

В обработчике предусмотрена проверка состояния флага InDOS и, если DOS свободна, запуск task, а если DOS занята – установка флага запроса запуска и выход из прерывания. В последнем случае повторные попытки активизации task осуществляются с помощью обработчика прерываний от таймера, который, обнаружив установленный флаг запроса запуска, в свою очередь, проверяет флаг InDOS и либо завершается (если DOS занята), либо запускает task (если DOS свободна). С целью упрощения программы в ней не проверяется состояние флага критической ошибки.

Для того чтобы аккуратно завершить задачу, используется обработчик прерываний по Ctrl+C и Ctrl+Break (вектор 23h). В этом обработчике выполняется единственная команда перехода на секцию завершения программы. В процессе завершения программы восстанавливаются векторы 08h и 09h и выполняется функция завершения 4Ch.

Пример 44.1. Использование средств DOS в обработчике аппаратных прерываний

```
text    segment 'code'
        assume  CS:text,DS:data
;Основная программа
myproc  proc
        mov     AX,data
        mov     DS,AX
;Выполним подготовительные действия
;Получим адрес флага InDOS
        mov     AH,34h
        int     21h
        mov     word ptr CS:in_dos,BX
        mov     word ptr CS:in_dos+2,ES
;Сохраним вектор прерывания от таймера 08h в ячейке old_08h
        mov     AX,3508h
        int     21h
        mov     word ptr CS:old_08h,BX
        mov     word ptr CS:old_08h+2,ES
;Сохраним вектор прерывания от клавиатуры (09h) в ячейке old_09h
        mov     AX,3509h
        int     21h
        mov     word ptr CS:old_09h,BX
        mov     word ptr CS:old_09h+2,ES
;Сохраним вектор прерывания по Ctrl+C (23h) в ячейке old_23h
        mov     AX,3509h
        int     21h
        mov     word ptr CS:old_09h,BX
        mov     word ptr CS:old_09h+2,ES
;Сохраним DS и настроим DS на сегмент команд
        push    DS
        push    CS
        pop     DS
;Заполним вектор 08h адресом нашего обработчика new_08h
        mov     AX,2508h
        mov     DX,offset new_08h
        int     21h
;Заполним вектор 09h адресом нашего обработчика new_09h
        mov     AX,2509h
        mov     DX,offset new_09h
        int     21h
;Заполним вектор 23h адресом нашего обработчика new_23h
        mov     AX,2523h
        mov     DX,offset new_23h
        int     21h
```

```
;Восстановим адресуемость сегмента команд
        pop    DS
;Будем периодически выводить сообщение
;Сначала line1 с возвратом курсора в начало строки
go:     mov    AH,40h         ;Файловая функция вывода
        mov    BX,1           ;Дескриптор экрана
        mov    CX,41          ;Число выводимых байтов
        mov    DX,offset line1;Адрес строки
        int    21h
        call   delay          ;Задержка
;Затем line2 с возвратом курсора в начало строки
        mov    AH,40h
        mov    CX,41
        mov    DX,offset line2
        int    21h
        call   delay          ;Задержка
        jmp    go             ;Бесконечный цикл
;Процедура task, активизируемая горячей клавишей
;и использующая функции DOS
task    proc
        push   AX             ;Сохраним все регистры,
        push   CX             ;используемые
        push   DX             ;в task
        push   DS             ;и в bin_dec
        mov    AX,data        ;Настроим DS на наш
        mov    DS,AX          ;сегмент данных
;Получим время с помощью функции DOS
        mov    AH,2Ch         ;Функция получения времени
        int    21h
;Преобразуем часы
        mov    AL,CH          ;Часы в AL
        mov    BX,offset time ;Адрес поля в BX
        call   bin_dec        ;Вызовем binasc
;Преобразуем минуты
        mov    AL,CL          ;Минуты в AL
        lea    BX,time+3      ;Адрес поля в BX
        call   bin_dec        ;Вызовем binasc
;Преобразуем секунды
        mov    AL,DH          ;Секунду в AL
        lea    BX,time+6      ;Адрес поля в BX
        call   bin_dec        ;Вызовем binasc
;Выведем всю строку функцией DOS
        mov    AH,09h
        mov    DX,offset string
        int    21h
        pop    ES             ;Восстановим
        pop    DX             ;регистры
        pop    CX
        pop    AX
        ret                   ;Вернемся в тот наш обработчик, откуда вызвана task
task    endp
home:
;Завершим программу, восстановив сначала все перехваченные векторы
;Восстановим вектор 09h
        mov    AX,2509h
        lds    DX,CS:old_09h
        int    21h
;Восстановим вектор 08h
        mov    AX,2508h
        lds    DX,CS:old_08h
        int    21h
```

```
;Завершим программу
        mov    AX,4C00h
        int    21h
myproc  endp
;Поля данных, размещенные для удобства адресации в сегменте команд
old_09h dd     0
old_08h dd     0
old_23h dd     0
in_dos  dd     0
task_req db    0
;Подпрограмма задержки
delay   proc
        mov    CX,0
del:    loop   del
        ret
delay   endp
;Подпрограмма преобразования двухразрядного двоичного числа
;в десятичное число в ASCII-представлении
;На входе       AL - число
;               DS:BX - адрес буфера для ASCII-представления
bin_dec proc
        xor    AH,AH          ;Очистим AH. Теперь число в AX
        div    ten            ;Разделим на 10. AL=десятки,
                              ;AH=единицы
        add    AL,'0'         ;Преобразуем AL в символьную форму
        mov    [BX],AL        ;Зашлем в строку
        add    AH,'0'         ;Преобразуем AH в символьную форму
        mov    1[BX],AH       ;Зашлем в строку
        ret
bin_dec endp
;Обработчик прерываний от клавиатуры
new_09h proc
        push   AX             ;Сохраним
        push   BX             ;используемые
        push   ES             ;регистры
        in     AL,60h         ;Введем скан-код
        cmp    AL,4Eh         ;"Серый плюс?"
        je     plus           ;Да, на продолжение
        pop    ES             ;Нет, восстановим
        pop    BX             ;сохраненные
        pop    AX             ;регистры
        jmp    CS:old_09h     ;и перейдем в системный обработчик
;Нажата клавиша "серый плюс"!
plus:   in     AL,61h         ;Выполним волшебные действия
        or     AL,80h         ;с контроллером клавиатуры,
        out    61h,AL         ;чтобы снять скан-код нажатой
        and    AL,7Fh         ;клавиши и разрешить
        out    61h,AL         ;дальнейшую работу контроллера
        mov    AL,20h         ;Пошлем в контроллер прерываний
        out    20h,AL         ;сигнал EOI
;Проверим, свободна ли DOS, и вызовем процедуру task
        les    BX,CS:in_dos;Получим адрес флага InDOS
        cmp    byte ptr ES:[BX],0;DOS свободна?
        jne    wait_dos       ;Нет, придется ждать
        call   task           ;Да, вызовем процедуру task
        jmp    out_09h        ;и на выход из обработчика
wait_dos:inc   byte ptr CS:task_req ;Установим флаг запроса запуска
out_09h:;mov   AL,20h         ;Пошлем в контроллер прерываний
        ;out   20h,AL         ;сигнал EOI
        pop    ES             ;Восстановим
        pop    BX             ;сохраненные
```

```
        pop     AX              ;регистры
        iret                    ;Возврат в прерванную программу
new_09h endp
;Обработчик прерываний от таймера
new_08h proc
        push    BX              ;Сохраним используемые
        push    ES              ;регистры
        pushf                   ;Перейдем в системный обработчик
        call    CS:old_08h      ;с возвратом
        cmp     CS:task_req,1;Флаг запроса запуска установлен?
        jne     out_08h         ;Нет, на выход из обработчика
        les     BX,CS:in_dos;Да, получим адреса флага InDOS
        cmp     byte ptr ES:[BX],0;Флаг InDOS сброшен?
        jne     out_08h         ;Нет, на выход из обработчика
        dec     CS:task_req     ;Да, сбросим флаг запроса запуска
        call    task            ;и вызовем task
out_08h:pop     ES              ;Восстановим сохраненные
        pop     BX              ;регистры
        iret                    ;и вернемся в прерванную программу
new_08h endp
;Обработчик прерываний по Ctrl+C и Ctrl+Break
new_23h proc
        jmp     home            ;На завершение программы
new_23h endp
text    ends
data    segment
;Поля данных, размещенные в сегменте данных
string  db      27,'[s',27,'[25;71H',27,'[34;41m'
time    db      '00:00:00'
        db      27,'[0m'
        db      27,'[u'
        db      '$'
ten     db      10
line1   db      40 dup ('-'),13
line2   db      40 dup ('|'),13
data    ends
stk     segment stack
        db      256 dup ('&')
stk     ends
        end     myproc
```

Главная процедура myproc начинается с получения и сохранения полного адреса системного флага InDOS (не самого флага!). Далее выполняется обычная процедура установки прикладных обработчиков прерываний: сначала чтение и сохранение используемых векторов 08h и 09h, а затем помещение в векторы 08h, 09h и 23h адресов прикладных обработчиков. Вектор 23h сохранять (и затем восстанавливать) нет необходимости, так как он будет восстановлен системой в процессе завершения задачи.

Выполнив инициализирующие действия, программа входит в бесконечный цикл попеременного вывода на экран двух строк, заполненных знаками – и |. Мерцание строки на экране свидетельствует о выполнении программой цикла вывода. Однако в таком цикле, в котором почти непрерывно выполняются программы DOS, практически невозможно дождаться освобождения DOS. В реальных программах помимо системных вызовов всегда есть "процессорные" участки, заполненные какими-либо командами процессора. Роль таких участков в приводимом примере выполняет подпрограмма delay, в которой команда loop выполняется 64 К раз, создавая небольшую задержку.

Алгоритмы проверки состояния флага InDOS как непосредственно в обработчике прерываний от клавиатуры, так и во вспомогательном обработчике прерываний от таймера не отличаются от описанных в начале этой статьи.

Некоторую сложность вызывает необходимость вывода разнородной информации на определенные места экрана. Для того чтобы исключить прокрутку экрана при выводе на него строк с символами – и |, строки заканчиваются кодом 13 возврата каретки (без кода 10 перевода строки). В результате обе строки выводятся на одно и то же место экрана, попеременно затирая друг друга. Строку же с текущим временем следует выводить на какое-то другое место. Для перемещения текстового курсора в приводимом примере используются Esc-последовательности: Esc[s – для сохранения текущего местоположения курсора, Esc[25,71H – для его позиционирования в правый нижний угол экрана, куда будет выводиться текущее время, и Esc[u – для восстановление сохраненной позиции курсора. Заодно для красоты изменяется атрибут выводимых символов. Таким образом, для правильного функционирования программы необходимо загрузить драйвер ANSI.SYS.

Отладьте программу и убедитесь в ее правильной работе. Нажатие клавиши "серый плюс" должно приводить к выводу в правый нижний угол экрана текущего времени. Нажатие Ctrl+C должно завершать программу.

Выполните исследования подготовленной программы. Рассмотрите реакцию программы на нажатие других клавиш. Обратите внимание на то, что после нажатия любой клавиши перестает работать Ctrl+C (программа не забирает коды из кольцевого буфера, и они маскируют введенную позже команду Ctrl+C). В этом случае используйте для завершения программы Ctrl+Break– эта комбинация обслуживается прикладной программой через тот же вектор 23h, но нажатие Ctrl+Break очищает кольцевой буфер клавиатуры и стирает ранее введенные в него символы (см. статью 40).

Исключите из обработчика прерываний от клавиатуры строки проверки флага InDOS:

```
;       cmp     byte ptr ES:[BX],0     ;DOS свободна?
;       jne     wait_dos     ;Нет, придется ждать
```

(строго говоря, достаточно исключить лишь вторую из приведенных выше строк). Убедитесь, что нажатие горячей клавиши приводит к разрушению DOS.

Устраните из процедуры task вызов функции 2Ch DOS (достаточно поставить знак комментария перед вызовом DOS int 21h). В этом случае программа не будет разрушать систему даже при выключении алгоритма ожидания освобождения DOS. Обратите внимание на то, что в этом случае нажатие клавиши "серый плюс" по-прежнему прерывает выполнение функции DOS 40h, а в обработчике прерываний от клавиатуры (конкретно – в процедуре task) по-прежнему вызывается функция DOS 09h. Однако функции 40h и 09h выполняются на разных стеках DOS ("дисковом" и ввода-вывода) и прерывание их друг другом вполне допустимо.

Статья 45. **Использование прерывания 28h**

Описанная в предыдущей статье методика годится далеко не для всех программ. Если текущая программа ждет ввода с клавиатуры, она не выходит из соответствующей функции DOS и флаг занятости DOS непрерывно установлен. То же получается, если текущей программы вообще нет, так как в этом случае активным является командный процессор COMMAND.COM, который ждет ввода с клавиатуры очередной

команды пользователя (функцией DOS 0Ah). В такой ситуации наш обработчик никогда не сможет вызвать требуемую функцию DOS. Для преодоления этого затруднения в DOS включено специальное прерывание int 28h, вызываемое функциями ввода с клавиатуры. Выполнение, например, функции 01h в самых общих чертах выглядит так, как показано на рис. 45.1.

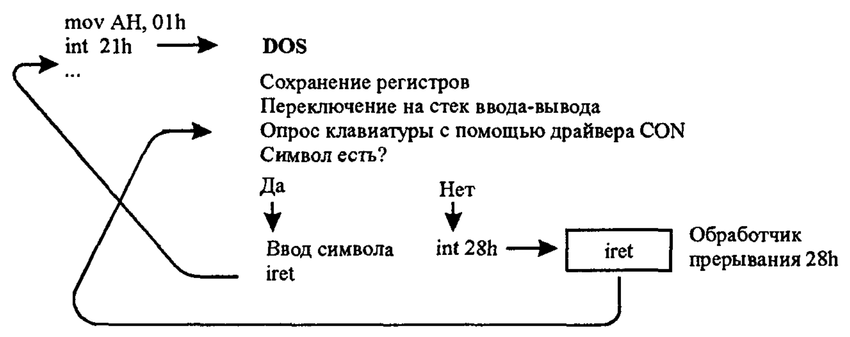


Рис. 45.1. Системный алгоритм выполнения функции ввода с клавиатуры

Системный обработчик прерывания 28h содержит единственную команду iret, поэтому вызов int 28h при выполнении функции ввода-вывода никак не нарушает хода программы. Однако прикладная программа может поместить в вектор 28h адрес своей процедуры обработки этого прерывания. Поскольку прерывание 28h возникает только при выполнении функций, использующих стек ввода-вывода, в обработчике прерывания 28h допустим вызов любых функций DOS из диапазона 0Dh...6Ch. Надо только иметь в виду, что при входе в обработчик 28h все регистры заполнены системными значениями и для выполнения прикладных функций их надо соответствующим образом инициализировать.

Таким образом, прикладной обработчик аппаратных прерываний, в котором вызываются функции DOS, должен включать, помимо активизатора по прерываниям от таймера (через вектор 08h), еще и активизатор по прерыванию 28h со схожим алгоритмом:

```
in_dos   dd     0             ;Адрес флага InDOS
crit_err dd     0             ;Адрес флага ErrorMode
task_req db     0             ;Флаг требования запуска task
old_28h  dd     0             ;Ячейка для хранения исходного вектора 28h
         ...
new_28h: cmp    CS:task_req,1;Процедура task требует запуска?
         jne    out_28h       ;Нет, можно завершить обработку
         les    BX,CS:in_dos;Да, получим адрес флага InDOS
         lds    SI,CS:crit_err;и адрес флага ErrorMode
         cmp    DS:[SI],0     ;Флаг ErrorMode сброшен?
         jne    out_28h       ;Нет, придется ждать
         cmp    ES:[BX],1     ;Флаг InDOS не больше 1?
         ja     out_28h       ;Больше, вложенный вызов DOS, вызов DOS запрещен
         dec    CS:task_req   ;Сбросим флаг требования запуска
         call   task          ;и вызовем task
out_28h  jmp    CS:old_28h    ;Завершим обработчик
```

Прикладной обработчик прерывания 28h (включенный в состав обработчика аппаратного прерывания) прежде всего проверяет, установлен ли флаг запроса запуска за-

дачи. Если этот флаг сброшен, надо просто вернуться в прерванную функцию DOS, для чего в простейшем случае можно выполнить команду iret. Если, однако, в системе имеется несколько обработчиков прерывания 28h, такое завершение лишит эти обработчики возможности фиксировать прерывание 28h. Поэтому наш обработчик лучше завершить командой jmp CS:old_28h передачи управления на тот обработчик, адрес которого мы вытеснили из вектора 28h.

Если флаг запроса запуска задачи установлен, сначала проверяется состояние флага критической ошибки и потом, если он сброшен, состояние флага занятости DOS. Этот флаг должен быть равен единице (ведь сейчас выполняется функция ввода с клавиатуры). Если значение флага больше единицы, это свидетельствует о том, что "внутри" прерывания 28h уже вызвана функция DOS и вызывать функции DOS, увеличивая тем самым уровень вложенности DOS, нельзя. Бывают ситуации (например, при использовании программы Norton Commander), когда в обработчике int 28h флаг InDOS оказывается равен не единице, а нулю. Проверка значения флага на условие больше единицы учитывает и эту возможность.

Наконец, после успешного прохождения всех проверок вызывается процедура task с вызовами DOS, после завершения которой управление передается предыдущему обработчику int 28h.

Вызов int 28h выполняется функцией DOS на стеке ввода-вывода. Поэтому в прикладном обработчике прерывания 28h можно вызывать только функции "дисковой" группы. Кроме того, недопустим вызов файловых функций с указанием стандартных дескрипторов клавиатуры или экрана (0...2). Каким же образом в таком обработчике можно выполнить ввод-вывод через терминал? Для этого следует "открыть" терминал как файл и полученный от DOS дескриптор использовать для операций ввода-вывода:

```
fname   db    'CON',0     ;Имя консоли в формате файловых функций
handle  dw    0           ;Ячейка для дескриптора консоли
        ...
        mov   AH,3Dh      ;Функция открытия файла
        mov   AL,2        ;Режим ввода и вывода
        mov   DX,offset fname;Адрес имени файла
        int   21h
        mov   handle,AX   ;Сохраним дескриптор консоли
;Введем сообщение с клавиатуры
        mov   AH,3Fh      ;Файловая функция ввода
        mov   BX,handle   ;Дескриптор консоли
        mov   CX,80       ;Предельное число вводимых символов
        mov   DX,offset buf_in;Адрес буфера
        int   21h
;Выведем сообщение
        mov   AH,40h      ;Файловая функция вывода
        mov   BX,handle   ;Дескриптор консоли
        mov   CX,mes_len  ;Длина сообщения
        mov   DX,offset mes;Адрес сообщения
        int   21h
```

Другой способ организовать ввод-вывод через терминал в обработчике прерывания 28h – воспользоваться функциями BIOS.

Модифицируем программу 44.1, заменив фрагмент вывода на экран строк line1 и line2 фрагментом ввода символа с клавиатуры функцией DOS 01h:

```
;Будем периодически вводить символы
go:     mov   AH,01h
        int   21h
        jmp   go
```

В такой редакции программы горячая клавиша "серый плюс" действовать не будет, так как программа почти непрерывно находится "внутри DOS" (убедитесь в этом экспериментально). Для того чтобы в такой ситуации обработчик аппаратных прерываний (не обязательно от клавиатуры или таймера) мог "прорваться" сквозь занятость DOS, в программу 44.1 следует добавить средства обработки прерывания 28h. Однако, как было показано выше, в обработчике прерывания 28h (точнее говоря, в нашей процедуре task, которая будет вызываться из обработчика new_28h) можно вызывать только функции DOS "дисковой" группы, в которых к тому же недопустимо использование стандартных дескрипторов консоли. Использованная же нами в примере 44.1 функция 09h принадлежит к группе ввода-вывода. Поэтому вывод на экран строки с текущим временем придется осуществить с помощью функции 40h, а дескриптор экрана получить, открыв консоль с помощью функции 3Dh, предназначенной для открывания файлов. В примере 45.1 приведены только добавляемые (или изменяемые) фрагменты нового варианта программы.

Пример 45.1. Обработка прерывания int 28h

```
;В секции инициализации:
;Сохраним вектор прерывания 28h в ячейке old_28h
        ...
;Заполним вектор 28h адресом обработчика new_28h
        ...
;В процедуре task после получения и преобразования текущего времени:
;Откроем терминал как файл и выведем строку
        mov     AH,3Dh          ;Откроем консоль как файл
        mov     AL,2            ;Режим ввода и вывода
        mov     DX,offset console;Адрес имени консоли
        int     21h
        mov     BX,AX           ;Сохраним полученный дескриптор
        mov     AH,40h          ;Выведем строку файловой функцией
        mov     DX,offset string
        mov     CX,34           ;Длина строки
        int     21h
;В секции обработчиков прерываний:
;Новый обработчик прерывания 28h
new_28h proc
        push    BX              ;Сохраним используемые
        push    ES              ;регистры
        cmp     CS:task_req,1;Флаг запроса запуска установлен?
        jne     out_28h         ;Нет, на выход
        les     BX,CS:in_dos;Да,получим адрес флага InDOS
        cmp     byte ptr ES:[BX],1;Флаг InDOS=1?
        jne     out_28h         ;Нет, на выход
        dec     CS:task_req ;Да, сбросим флаг запроса запуска
        call    task            ;и вызовем процедуру task
out_28h:pop     ES              ;Восстановим
        pop     BX              ;регистры
        jmp     CS:old_28h  ;Перейдем в системный обработчик int 28h
new_28h endp
;Будем периодически вводить сообщение
go:     mov     AH,01h
        int     21h
        jmp     go
;В секции завершения:
;Восстановим вектор 28h
        ...
```

Статья 46. **Взаимодействие программы с файловой системой**

Программные обращения к файлам (создание, удаление, чтение, запись и другие операции) являются почти непременным атрибутом любой прикладной программы. Обычно для работы с файлами используются предназначенные для этого функции DOS, вызов которых весьма прост и не требует знания деталей организации файловой системой. Однако для ряда применений (защита программ от несанкционированного копирования, обращение к файлам из резидентных программ и др.) необходимо более детально представлять себе механизм взаимодействия программы с файловой системой и состав используемых при этом системных полей данных.

Операционная система MS-DOS использует в своей работе целый ряд системных таблиц, лишь частично нашедших отражение в официальной документации. Пользоваться этими средствами следует с осторожностью, так как их форматы могут различаться в разных версиях DOS, однако обращение к ним оказывается во многих случаях необходимым. Ниже будут рассмотрены наиболее важные системные таблицы, имеющие отношение к обслуживанию дисков и файлов.

Основополагающей системной таблицей, содержащей адреса целого ряда системных структур, является так называемый список списков (List of lists). Адрес списка списков можно получить, используя функцию 52h прерывания 21h. Функция возвращает двухсловный адрес списка списков в регистрах ES:BX. В табл. 46.1 приведены выборочные данные из списка списков.

Таблица 46.1. Содержимое некоторых полей списка списков

Смещение	*Число байтов*	*Описание*
-8	4	Адрес текущего буфера диска
-2	2	Сегмент первого блока управления памятью
00h	4	Адрес первого блока параметров диска
04h	4	Адрес первой системной таблицы файлов SFT
08h	4	Адрес заголовка активного устройства CLOCK$
0Ch	4	Адрес заголовка активного устройства CON
10h	2	Максимальный размер сектора в байтах
16h	4	Адрес массива структур текущего каталога
20h	1	Число установленный блочных устройств
21h	1	Число возможных букв обозначения дисков
22h	18	Заголовок драйвера устройства NUL
3Fh	2	Значение x в директиве BUFFERS=x,y
41h	2	Значение y в директиве BUFFERS=x,y
43h	1	Дисковод загрузки (1=A:)
45h	2	Размер расширенной памяти в килобайтах

В байтах 4...7 списка списков хранится адрес системной таблицы файлов (System file table, SFT), куда DOS записывает характеристики открываемых по заказам выполняемой программы файлов или стандартных устройств, к которым предполагается доступ средствами файловой системы. По умолчанию при загрузке DOS формируется одна SFT, в которую входит 5 блоков описания файлов.

В такой конфигурации компьютер практически не может эксплуатироваться, так как три блока система занимает под характеристики стандартных устройств CON, AUX и PRN и для открываемых программами файлов остается всего два блока. При включении в файл CONFIG.SIS директивы BUFFERS DOS создает вторую таблицу, связывая ее с первой (рис. 46.1). Суммарное число блоков в двух таблицах равно первому параметру директивы BUFFERS.

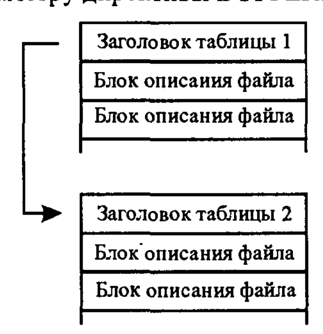


Рис. 46.1 Системные таблицы файлов

Каждая SFT начинается с 6-байтового заголовка (табл. 46.2).

Таблица 46.2. Заголовок SFT

Смещение	*Число байтов*	*Описание*
0	4	Адрес следующей таблицы или FFFFh в 1-м слове
4	2	Число блоков описания файлов в данной таблице

Далее идут блоки описания файлов в количестве, указанном в заголовке таблицы. Формат блоков и даже их размер зависят от версии DOS. В табл. 46.3 приведены наиболее интересные для прикладного программиста поля блока описания файла для DOS начиная с версии 4.

Таблица 46.3. Блок описания файла

Смещение	*Число байтов*	*Описание*
00h	2	Число дескрипторов, закрепленных за данным файлом, или 0, если данный блок свободен
02h	2	Режим доступа к файлу, заданный при его открытии
04h	1	Атрибуты файла
05h	2	Информационное слово устройства
07h	4	Адрес заголовка драйвера символьного устройства или (для файлов) адрес блока параметров дисковода
0Bh	2	Номер первого кластера файла
0Dh	2	Дата последней модификации файла
0Fh	2	Время последней модификации файла
11h	4	Размер файла в байтах
15h	4	Текущее положение указателя в файле
19h	2	Относительный номер последнего кластера файла, к которому выполнялось обращение
1Bh	4	Номер сектора с записью каталога о данном файле

1Fh	1	Номер записи каталога внутри сектора
20h	11	Имя и расширение файла
31h	2	Сегментный адрес PSP программы, открывшей файл (ID программы)
35h	2	Абсолютный номер последнего кластера файла, к которому выполнялось обращение

Как видно из табл. 46.3, часть информации, составляющей блок описания открытого файла, поступает из записи каталога. Это относится к таким полям, как имя файла, его длина, номер первого кластера файла, и др. Все эти данные программа может извлечь как из каталога диска, так и из SFT. Однако поиск на диске каталога, содержащего запись о требуемом файле, является весьма трудоемким делом. Найти SFT гораздо легче. К тому же в SFT содержатся дополнительные сведения. Так, с помощью SFT можно найти драйвер данного устройства или проследить расположение файла на диске, если читать файл кластер за кластером и анализировать поле SFT со смещением 35h.

При обращении к SFT DOS пользуется относительными номерами блоков описания файлов. Нумерация блоков начинается с нуля и идет насквозь через обе таблицы.

Поскольку порядок описания открытых файлов в SFT заранее неизвестен, для нахождения интересующего нас блока описания файла следует открыть файл функцией 3Dh, получить от системы его дескриптор, а затем по дескриптору найти соответствующий ему блок описания файла в SFT. Для выполнения этой операции надо знать, как значение дескриптора связано с местоположением блока описания файла в SFT.

В префиксе программы (PSP) имеется таблица, называемая таблицей файлов задания (Job File Table, JFT). По умолчанию эта таблица начинается с байта 18h PSP и имеет размер 20 байт. В байты JFT DOS по мере открытия файлов записывает номера соответствующих блоков описания файлов в SFT. Свободные байты JFT содержат коды FFh. Относительные номера байтов JFT (от ее начала) и служат дескрипторами открываемых файлов (рис. 46.2).

Первые 5 дескрипторов (от 0 до 4) система закрепляет за стандартными устройствами CON, AUX и PRN. Этим устройствам соответствуют первые три блока описания файлов в первой SFT. Таким образом, первый свободный дескриптор должен иметь значение 5, а соответствующая ему информация о файле располагается в блоке описания файла с номером 3. Практически после загрузки компьютера в SFT может оказаться занято больше блоков, так как при наличии в файле AUTOEXEC.BAT команд загрузки резидентных программ с перенаправлением на нуль-устройство вида

```
C:\TOOLS\PU_1700.COM > NUL
```

в SFT отводятся блоки под описание устройства NUL.

На рис. 46.2 изображены обе таблицы файлов в ситуации, когда в файле CONFIG.SYS присутствует директива

```
FILES=12
```

и текущая программа открыла два файла. DOS на этапе конфигурирования создала вторую SFT объемом 7 блоков. При открытии текущей программой двух файлов DOS отвела в SFT под характеристики этих файлов блоки 3 и 4 и поместила номера этих блоков в первые свободные байты JFT с номерами 5 и 6. Таким образом, файлы получили дескрипторы со значениями 5 и 6.

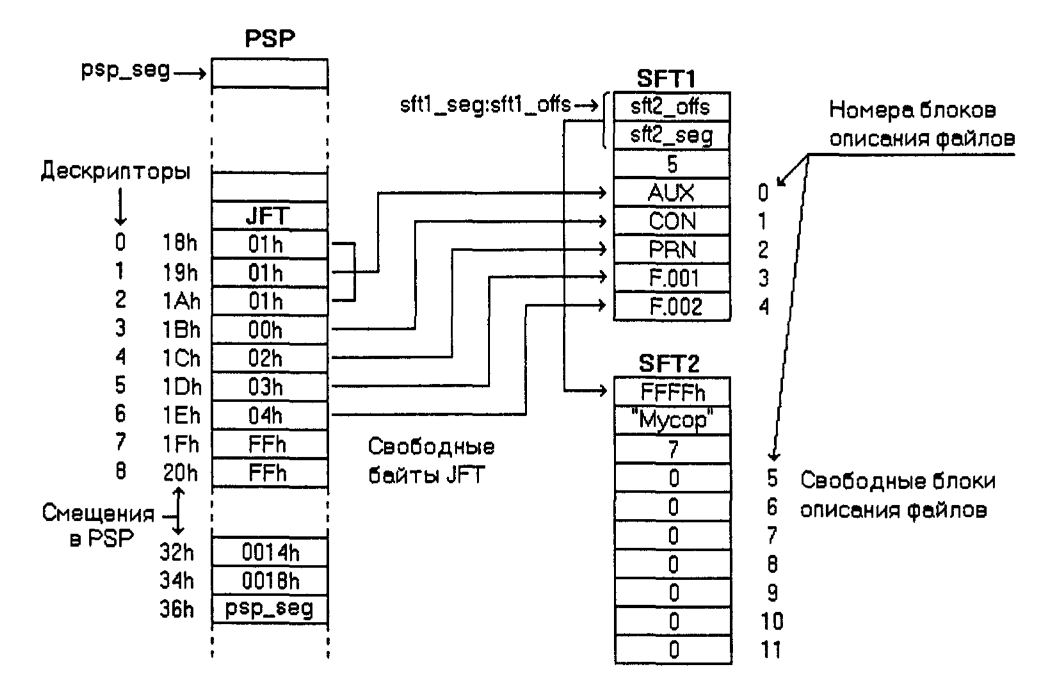


Рис. 46.2. Таблица файлов задания JFT и системная таблица файлов SFT

В PSP программы, кроме самой таблицы файлов задания, имеются еще служебные поля, связанные с этой таблицей. В слове со смещением 32h от начала PSP хранится размер JFT (по умолчанию 20=14h), а в двухсловной ячейке со смещением 34h записан полный двухсловный адрес JFT. По умолчанию в слове со смещением 34h записано число 18h, а в слове со смещением 36h – сегментный адрес PSP, который при загрузке программы поступает также в регистры DS и ES.

Поскольку размер JFT по умолчанию составляет 20 байт, и при этом 5 полей JFT заняты предопределенными дескрипторами, программа может открыть не более 15 файлов, при условии, что в файл CONFIG.SYS включена директива FILES с достаточно большим значением параметра (не меньше 20). Для того чтобы получить возможность открыть большее число файлов, программа должна с помощью системной функции 67h создать новую JFT. DOS, выделив для нее память, запишет в соответствующие поля PSP новые адрес и размер этой таблицы, скопирует содержимое старой JFT в новую таблицу и впредь будет открывать файлы с помощью этой новой JFT.

Теперь нетрудно сообразить, что происходит при запуске некоторой программы с перенаправлением ее ввода или вывода. Командный процессор COMMAND.COM, получив с клавиатуры команду пользователя вида

```
PROG.EXE > FILE001.DAT
```

создает на диске (в данном случае в текущем каталоге текущего диска) указанный в команде файл FILE001.DAT и помещает номер отведенного ему в SFT блока не в первый свободный байт JFT, а в байт, отвечающий дескриптору 1 (дескриптор стандартного вывода). После этого весь вывод программы, выполняемый с помощью файловых функций вывода через дескриптор 1, будет поступать не на экран, а в файл

FILE001.DAT. Точно так же перенаправляется стандартный ввод; в этом случае модифицируется байт JFT, закрепленный за дескриптором 0.

Рассмотрим в качестве примера работы с SFT простой вариант методики защиты загрузочного модуля программы от несанкционированного копирования. Идея защиты заключается в том, что программа привязывается к номеру кластера, с которого начинается выполнимый файл программы на диске. Привязка осуществляется следующим образом. Специально подготовленная установочная программа открывает файл с рабочей программой и по таблице открытых файлов находит начальный номер кластера. Это число, являющееся своеобразным ключом, записывается установочной программой в определенное место файла рабочей программы. Рабочая же программа после запуска прежде всего выполняет ту же операцию – определяет свой начальный адрес, а затем сравнивает его с ключом. Если числа совпадают, программа приступает к выполнению своей содержательной части; если не совпадают – аварийно завершается. При копировании программы на другой диск (или даже на тот же самый) она окажется расположенной в другом месте и номер кластера, записанный установочной программой, уже не будет соответствовать реальному адресу файла. В то же время с помощью установочной дискеты (очевидно, не входящей в комплект поставки программы потребителю) программу нетрудно установить на любом диске.

Номер первого кластера программы можно получить из записи каталога, но проще найти его в таблице открытых файлов.

Некоторая проблема возникает с поиском в файле программы ячейки с ключом, в данном случае – номером первого кластера. Проще всего расположить эту ячейку в известном месте сегмента команд, например в его первом слове, а расположение сегмента команд относительно начала программы найти из заголовка файла .EXE (см. табл. 32.2). В примерах 46.1 и 46.2 приведены тексты обеих программ (рабочей и установочной) с краткими комментариями.

Пример 46.1. Защита программы от копирования. Рабочая (защищаемая) программа

```
text     segment
         assume cs:text,ds:data
mark     dw     9999h       ;Поле для номера первого кластера
main     proc
begin:   mov    AX,data     ;Настроим DS
         mov    DS,AX       ;на сегмент данных
         call   search      ;AX=номер первого кластера
         cmp    AX,CS:mark  ;Сравним с записанным в программе
         jne    notins      ;Не равны, на завершение
         mov    AH,09h      ;Равны, выведем сообщение
         mov    DX,offset okey;и приступим к содержательной части программы
         int    21h         ;(здесь отсутствует)
         ...                ;Содержательная часть программы
         jmp    outprog     ;Переход на завершение
notins:  mov    AH,09h      ;Выведем сообщение о том, что
         mov    DX,offset notokey;данная копия не установлена
         int    21h
outprog:mov     AX,4C00h    ;Завершение программы
         int    21h
main     endp
;Процедура определения номера первого кластера, занимаемого
;файлом с программой на диске. Номер кластера возвращается в AX
search   proc
;Откроем файл с этой программой, чтобы он попал в таблицу файлов
         mov    AH,3Dh      ;Функция 3Dh открытия файла
         mov    AL,02h      ;Доступ для чтения-записи
         mov    DX,offset myname;Поле с именем файла
```

```
        int     21h         ;AX=дескриптор открытого файла
        mov     handle,AX   ;Сохраним дескриптор
;Найдем в JFT номер блока SFT с описанием этого файла
        mov     DI,18h      ;ES:DI→JFT в PSP
        add     DI,AX       ;ES:DI→дескриптор этого файла
        mov     CL,ES:[DI]  ;CL=индекс в SFT
        xor     CH,CH       ;CX=индекс в SFT
;Получим доступ к системной таблице файлов SFT
        mov     AH,52h
        int     21h
        les     DI,ES:[BX+4];ES:DI→первая SFT
        cmp     CX,ES:[DI+4];Индекс в первой SFT?
        jb      here        ;Да
        sub     CX,ES:[DI+4];Нет, вычтем число блоков в этой SFT
        les     DI,ES:[DI]  ;ES:DI→вторая SFT
;Нашли ту SFT, в которой блок этого файла
here:   add     DI,6        ;ES:DI→первый блок описания файла в SFT
        mov     AX,59       ;Размер блока описания файла
        mul     CL          ;Умножим на индекс
        add     DI,AX       ;ES:DI→блок описания этого файла
        mov     AX,ES:[DI+0Bh];AX=номер первого кластера файла
        ret
search  endp
text    ends
data    segment
myname  db      'work.exe',0;Пусть наша программа называется WORK.EXE
handle  dw      0           ;Ячейка для дескриптора открытого файла
okey    db      'Доступ к программе разрешен$'
notokey db      'Несанкционированная копия. Программа завершается$'
data    ends
```

Операции по определению номера начального кластера файла вынесены в процедуру search, которая используется и в рабочей, и в установочной программах. Легко заметить, что в рабочей программе нет необходимости сохранять полученный в этой процедуре дескриптор файла, так как программа далее к нему не обращается. В установочной же программе этот дескриптор используется еще неоднократно. Мы оставили строки (и ячейку) его сохранения, чтобы не вносить различий в процедуры search той и другой программ.

Пример 46.2. Защита программы от копирования. Установочная программа

```
text    segment
        assume cs:text,ds:data
main    proc
        mov     AX,data
        mov     DS,AX
;Откроем файл с рабочей программой и по содержимому SFT
;найдем номер его первого кластера (он возвращается в AX)
        call    search
        mov     cluster,AX  ;Сохраним его
;Файл с рабочей программой уже открыт. Прочитаем его заголовок
        mov     AH,3Fh
        mov     BX,handle
        mov     CX,512
        mov     DX,offset header
        int     21h
;Установим указатель в файле с рабочей программой на поле для номера кластера
        mov     DX,header+08h;DX=размер заголовка в параграфах
        add     DX,header+16h;Добавим смещение сегмента команд
                             ;от начала программы в параграфах
```

```
        mov     CL,4            ;Умножим на 16, чтобы перевести в байты
        shl     DX,CL           ;DX=Смещение сегмента команд от начала файла .EXE
                                ;Наш ключ начинается с байта 0 сегмента команд
        mov     AH,42h          ;Функция установки указателя в файле
        mov     AL,0            ;Режим задания указателя - от начала
        mov     BX,handle       ;Дескриптор
        mov     CX,0            ;Старшая часть указателя (в DX уже есть младшая)
        int     21h
;Запишем в поле для кластера найденный ранее номер первого кластера
        mov     AH,40h
        mov     BX,handle
        mov     CX,2
        mov     DX,offset cluster
        int     21h
;Выведем сообщение
        mov     AH,09h
        mov     DX,offset mesg
        int     21h
        mov     AX,4C00h
        int     21h
main    endp
search  proc
        ...     ;Текст процедуры см.в примере 46.1
search  endp
text    ends
data    segment
myname  db      'work.exe',0
handle  dw      0               ;Ячейка для дескриптора файла
cluster dw      0               ;Найденный первый кластер файла
header  dw      256 dup (0)
mesg    db      'Программа WORK.EXE установлена$'
data    ends
stk     segment stack 'stack'
        db      128 dup (?)
stk     ends
        end     main
```

Выше уже отмечалось, что программы реального режима, использующие тонкие системные или аппаратные возможности компьютера, могут неправильно работать в сеансе DOS систем Windows. Это относится и к рассмотренным примерам. Хотя DOS систем Windows 95/98 и создает таблицы открытых файлов, однако использует их не совсем так, как исходная система MS-DOS (например, версии 6.22). Так, в этих таблицах не заполняются поля с именем открываемого файла и с номерами кластеров. В результате рассмотренный программный комплекс в сеансе DOS систем Windows работать не будет.

Раздел четвертый

ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ СИСТЕМНЫХ СРЕДСТВ

Статья 47. **Запись и чтение файлов**

Как уже отмечалось ранее, в машинах типа IBM PC предусмотрены два уровня обращения к магнитным дискам. При работе на нижнем уровне пользователь с помощью прерывания BIOS INT 13h обращается непосредственно к программам управления диском. Типичными операциями этого уровня являются запись или чтение секторов или форматирование дорожки. Файловая система DOS не используется; требуемая информация отыскивается не по имени файла, а по номерам поверхности, цилиндра и сектора.

Верхний уровень реализуется с помощью прерывания DOS INT 21h, поддерживающего, наряду с прочими, также и функции обслуживания файловой структуры. Программист работает не с программами управления физическим диском, а с файловой системой DOS, получая возможность оперировать такими понятиями файловой системы, как логический диск, каталог, файл.

Как известно, для удобства работы с большим количеством разнородных файлов в DOS используется древовидная структура каталогов. Каталог представляет собой файл, обычно относительно небольшого размера, в котором содержится перечень всех подкаталогов следующего уровня и файлов, входящих в данный каталог. Каждому подкаталогу или файлу в каталоге отводится один элемент размером 32 байта, в который DOS заносит информацию о файле: имя, начальный адрес на диске (номер кластера), дату и время создания, длину в байтах, а также набор характеристик файла, называемых его атрибутами. Кроме элементов, относящихся к нижележащим каталогам и файлам, каждый каталог содержит еще два элемента: о себе самом и о родительском каталоге. Формат элемента каталога был приведен на рис. 29.2.

При создании нового файла DOS сама отыскивает на диске свободное место и назначает его новому файлу, создавая новый элемент в каталоге и заполняя его соответствующей этому файлу информацией. Хотя минимальной порцией информации, передаваемой контроллером диска в процессе записи или чтения файла, является сектор, и программы BIOS работают как раз с секторами, файловая система назначает место на диске целыми кластерами. Размер кластера на дискете составляет один сектор (512 байт); на жестком диске в кластер могут входить 4...8 секторов. Таким образом, минимальный физический размер файла, даже если данные в нем занимают лишь несколько байтов, составляет один кластер. Однако в элементе каталога указывается не физическая, а логическая длина файла, т. е. объем содержащихся в нем данных в байтах.

Методика работы с файлами существенно определяется тем обстоятельством, что каждый файл может занимать на диске несколько несмежных областей, т. е. быть разрывным. Такая система выделения дискового пространства позволяет, во-первых, в процессе работы с файлом многократно дописывать в него новые данные, увеличивая при этом длину файла, и, во-вторых, снимает проблемы с фрагментацией диска, поскольку даже самые маленькие и разрозненные свободные области на диске могут быть использованы для размещения нового файла. Следует, однако, иметь в виду, что сильно фрагментированный файл требует заметно больше времени для чтения или записи, что снижает скорость выполнения работающей с ним программы.

Работа с файлами предполагает использование дескрипторов (файловых индексов), которые в первом приближении можно рассматривать как номера открытых файлов.

Процедура чтения-записи файла в общем случае распадается на следующие операции:

- создание файла с заданным именем в указанном каталоге или открытие файла, если он был создан ранее;
- запись в файл или чтение из файла всего содержимого либо любой его части;
- закрытие файла.

В большинстве случаев работа с файлом начинается с выполнения операции его открытия, для чего предусмотрена особая функция DOS. Открывая файл, DOS назначает ему очередной свободный блок описания файла в системной таблице открытых файлов (System File Table, SFT), о которой шла речь в предыдущей статье. Как уже отмечалось, объем этой таблицы, определяющий максимальное число файлов, с которыми можно работать одновременно, задается на этапе конфигурирования DOS директивой FILES файла CONFIG.SYS.

Найдя в системе каталогов диска элемент, описывающий открываемый файл, DOS заносит в выделенный ему блок SFT основные характеристики файла, такие, как имя, длина, атрибуты, дата и время создания, стартовый кластер, физический адрес на диске элемента каталога, содержащего информацию о файле, и ряд других. Часть информации переписывается в блок SFT из элемента каталога, часть (например, указатель на блок параметров диска, где хранится информация о физических характеристиках диска) DOS поставляет сама. Важным элементом блока описания файла является двухсловная ячейка, в которой хранится указатель – номер байта относительно начала файла, с которого начнется очередная операция записи или чтения. Наличие указателя позволяет организовать прямой доступ к файлу, т. е. чтение или запись начиная с любого места файла. Ссылку на номер выделенного файлу блока в SFT DOS возвращает в программу в виде дескриптора.

Обращение к открытому файлу (запись, чтение, изменение характеристик файла и т. д.) осуществляется по присвоенному ему дескриптору; неоткрытый файл дескриптора не имеет, и система работать с ним не может. По мере выполнения операций с открытым файлом DOS модифицирует информацию в блоке SFT; содержимое SFT всегда отражает текущее состояние файла.

После окончания работы с файлом его надо закрыть предназначенной для этого функцией DOS. В процессе закрытия осуществляется сброс на диск буферов DOS, модификация элемента каталога и освобождение блока описания файла в SFT вместе с закрепленным за ним дескриптором. И то и другое можно теперь использовать для работы с другим файлом. Таким образом, система может последовательно работать с не-

ограниченным количеством файлов, но число одновременно открытых файлов определяется объемом системной таблицы файлов.

При завершении программы (для этого предусмотрена функция DOS 4Ch) выполняется автоматическое закрытие всех открытых в программе файлов. Поэтому в простых и не слишком ответственных программах файлы можно явным образом не закрывать – они все равно будут закрыты системой.

Рассмотрим несколько примеров операций записи и чтения файлов на диске.

Пример 47.1. Создание файла и запись в него данных

```
;В сегменте команд
;Создадим файл
        mov    AH,3Ch         ;Функция создания файла
        mov    CX,0           ;Без атрибутов
        mov    DX,offset fname;Адрес имени файла
        int    21h
        mov    handle,AX      ;Сохраним дескриптор файла
;Запишем в файл данные (в данном примере - текстовую строку)
        mov    AH,40h         ;Функция записи в файл
        mov    BX,handle      ;Дескриптор
        mov    CX,buflen      ;Число записываемых байтов
        mov    DX,offset bufout;Адрес данных
        int    21h
;Закроем файл (нет необходимости)
        mov    AH,3Eh         ;Функция закрытия файла
        mov    BX,handle      ;Дескриптор
        int    21h
;В сегменте данных
bufout  db 'Файл номер 1'  ;Данные для записи в файл
buflen=$-bufout            ;Ее длина (12 байт)
handle  dw 0               ;Ячейка для дескриптора
fname   db 'MYFILE.001',0  ;Имя файла в формате ASCIIZ
```

Функция 3Ch позволяет создать на диске файл с заданным именем. Спецификация файла, т. е. путь к нему вместе с именем файла и его расширением, указывается в виде символьной строки, завершающейся двоичным нулем ("строка ASCIIZ"). Для спецификации файла в программе действуют обычные правила DOS. Так, если в спецификации отсутствует путь, файл создается в текущем каталоге текущего диска (как в вышеприведенном примере); если указаны путь и имя файла, он создается в соответствующем каталоге текущего диска и т. д.

Если функция 3Ch обнаруживает, что на диске уже имеется файл с указанным именем, она фактически уничтожает его и создает новый с тем же именем. Поэтому пример 47.1 можно выполнять многократно; при каждом прогоне программы файл MYFILE.001 будет создаваться заново.

Функция 40h позволяет записывать данные на любое устройство, в том числе и в файл на диске. Конкретный приемник данных задается его дескриптором. Следует заметить, что при записи в файл, как, впрочем, и при выводе на любое устройство, в приемник данных поступают лишь те данные, которые указаны в программе. Никакие дополнительные коды, например обозначающие конец файла, не записываются. Таким образом, созданный в примере 47.1 файл будет иметь длину точно 12 байт (хотя фактически займет на диске целый кластер, в котором после наших данных будет "мусор").

В качестве данных, записываемых в файл, в примере выбрана текстовая строка. В этом случае легко прочитать файл с помощью любого текстового редактора и проконтролировать правильность выполнения программы. В действительности в файл

можно выводить любые данные. Функция записи 40h (как и функция чтения 3Fh, которая будет использована в примере 47.2) рассматривает пересылаемые данные просто как последовательность байтов, никак не анализируя их значения.

Пример 47.2. Чтение файла

```
;В сегменте команд
;Откроем файл
        mov    AH,3Dh            ;Функция открытия файла
        mov    AL,2              ;Доступ для чтения-записи
        mov    DX,offset fname   ;Адрес имени файла
        int    21h
        mov    handle,AX         ;Сохраним дескриптор
;Поставим запрос на чтение 80 байт
        mov    AH,3Fh            ;Функция чтения
        mov    BX,handle         ;Дескриптор
        mov    CX,80             ;Столько читать
        mov    DX,offset bufin   ;Сюда
        int    21h
        mov    CX,AX             ;Столько реально прочитали
;Выведем прочитанное на экран
        mov    AH,40h            ;Функция записи
        mov    BX,1              ;Дескриптор стандартного вывода
        mov    DX,offset bufin   ;Отсюда выводить (CX байт)
        int    21h
;В сегменте данных
bufin   db 80 dup (' ')          ;Вуфер ввода
handle  dw 0                     ;Ячейка для дескриптора
fname   db 'MYFILE.001',0        ;Имя файла в формате ASCIIZ
```

Функция 3Dh позволяет открыть уже имеющийся файл. В регистрах DS:DX задается адрес спецификации файла в виде строки ASCIIZ; в регистре AL – режим доступа (0 – чтение, 1 – запись, 2 – чтение и запись). Функция возвращает дескриптор открытого файла в регистре AX.

Чтение файла осуществляется вызовом функции 3Fh, которая требует указания в качестве входных параметров дескриптора источника данных (в регистре BX), адреса приемного буфера (в регистрах DS:BX) и количества передаваемых байтов (в регистре CX). Если мы хотим прочитать все содержимое файла, но не знаем точно его длину, можно в запросе на чтение указать заведомо большее число байтов (не более 65 535). Функция 3Fh сама определит длину файла и прочитает все его содержимое до конца. После возврата из DOS в регистре CX будет содержаться число фактически прочитанных байтов. В примере 47.2 содержимое AX переносится в CX и используется затем в качестве параметра для функции 40h, с помощью которой прочитанные данные выводятся на экран для контроля.

DOS предоставляет возможность обращения к любому байту файла по его номеру. С этой целью для каждого открытого файла DOS создает и поддерживает указатель (хранящийся в SFT), который представляет собой относительный номер байта в файле, начиная от которого будут выполняться запись или чтение данных. Указатель только что открытого или созданного файла позиционируется системой на начало файла, а функции чтения или записи смещают его на число прочитанных или записанных байтов. Таким образом, повторное использование функций чтения или записи реализует последовательный доступ к файлу. Для организации прямого доступа к произвольному месту файла предусмотрена функция 42h, позволяющая задать положение указателя относительно начала файла (для этого надо задать AL=0), конца файла (AL=2) или те-

кущего положения указателя (AL=1). Само значение смещения указателя (со знаком) заносится в регистры CX (старшая половина) и DX (младшая). В примере 47.3 проиллюстрирована методика прямого доступа к файлу.

Пример 47.3. Прямой доступ к файлу

```
;В сегменте команд
;Откроем файл
        mov    AH,3Dh         ;Функция открытия файла
        mov    AL,2           ;Доступ для чтения-записи
        mov    DX,offset fname;Адрес имени файла
        int    21h
        mov    handle,AX      ;Сохраним дескриптор
;Установим указатель на байт номер 11
        mov    AH,42h         ;Функция установки указателя
        mov    AL,0           ;Режим - от начала файла
        mov    BX,handle      ;Дескриптор
        mov    CX,0           ;Старшая половина указателя
        mov    DX,11          ;Младшая половина указателя
        int    21h
;Модифицируем файл
        mov    AH,40h         ;Функция записи
        mov    BX,handle      ;Дескриптор
        mov    CX,2           ;Число записываемых байтов
        mov    DX,offset mod;Отсюда выводить
        int    21h
;В сегменте данных
mod     db '2a'               ;Буфер вывода
handle  dw 0                  ;Ячейка для дескриптора
fname   db 'MYFILE.001',0 ;Имя файла в формате ASCIIZ
```

В приведенном примере предполагается, что модификации подлежит созданный ранее файл MYFILE.001, содержащий строку

Файл номер 1

в которой следует заменить 1 на 2a (что требует, кстати, как замены последнего байта файла, так и увеличения длины файла на 1 байт).

Открыв файл и получив его дескриптор, можно вызвать функцию 42h для установки указателя в файле. В параметрах этой функции мы указываем режим установки указателя (от начала файла) и 32-битовое значение смещения в файле (в нашем случае 11). После этого вызовом функции 40h выполняется запись в файл 2 байт из программного буфера mod.

В приведенном примере, где новый вывод в файл требуется выполнить от его последнего байта, было бы проще задать положение указателя от конца файла. Для этого в параметрах функции 42h следует указать режим 02, а в регистровую пару CX:DX поместить число –1 (так как мы хотим сместиться на 1 байт влево от конца файла); 32-разрядное число –1 имеет машинное представление FFFFFFFFh. Поэтому строки заполнения регистров CX и DX будут выглядеть следующим образом:

```
        mov    CX,0FFFFh
        mov    DX,0FFFFh
```

Если программист затрудняется в определении машинного представления отрицательного числа, можно предусмотреть в полях данных двухсловную ячейку для указателя, указав ее содержимое в десятичной форме:

```
pointer dd     -1
```

В процессе трансляции для числа –1 будет найдено и записано в ячейку pointer его машинное представление. В этом случае для заполнения регистров CX и DX потребуются следующие программные строки:

```
        mov    CX,word ptr pointer+2;Старшая часть числа
        mov    DX,word ptr pointer;Младшая часть числа
```

Статья 48. **Изменение характеристик файлов**

DOS предоставляет набор функций для изменения таких характеристик файла, как имя, атрибуты, время и дата создания. Первые две функции, работающие, в сущности, не с самим файлом, а с записью каталога, не требуют предварительного открытия файла и получения его дескриптора. В числе параметров последней функции необходимо указать дескриптор файла, т. е. файл сначала нужно открыть. Рассмотрим в качестве примера программу задания даты и времени создания файла (пример 48.1). Такая операция может понадобиться в процессе подготовки большого программного пакета. Часто всем файлам, составляющим пакет, назначают перед его выпуском одинаковое время создания, хотя реально эти файлы могли разрабатываться в течение длительного времени.

Пример 48.1. Назначение файлу даты и времени создания

```
;В сегменте команд
;Откроем файл
        mov    AH,3Dh          ;Функция открытия файла
        mov    AL,2
        mov    DX,offset fname ;Адрес имени файла
        int    21h
        mov    handle,AX       ;Получили дескриптор
;Изменим дату и время создания файла
        mov    AH,57h          ;Функция даты/времени
        mov    AL,1            ;Установить дату/время
        mov    BX,handle       ;Дескриптор файла
        mov    CX,0            ;Очистим CX
        or     CX,sec          ;Добавим секунды
        or     CX,min          ;Добавим минуты
        or     CX,hour         ;Добавим часы
        xor    DX,DX           ;Очистим DX
        or     DX,day          ;Добавим день
        or     DX,month        ;Добавим месяц
        or     DX,year         ;Добавим год
        int    21h
;В сегменте данных
handle  dw     0               ;Ячейка для дескриптора
fname   db 'MYFILE.001',0      ;Имя файла в формате ASCIIZ
sec     dw     6/2             ;6 секунд
min     dw     15*32           ;15 минут
hour    dw     12*2048         ;12 часов
day     dw     10              ;10 число
month   dw     3*32            ;Март
year    dw     21*512          ;21 год от 1980, т. е. 2001 г.
```

Для установки даты и времени создания файла используется функция 5701h. Она требует указания даты (в регистре DX) и времени (в регистре CX) в том же формате, в котором эти данные хранятся в элементе каталога (см. табл. 29.3). Формирование данных осуществляется непосредственно в регистрах с помощью нескольких команд по-

битового сложения or, которые позволяют заполнять группы битов слова новым содержимым, не затрагивая уже заполненных битов. Сдвиг составляющих даты/времени в соответствующие места осуществляется с помощью команд умножения на этапе трансляции. Например, для сдвига числа минут в группу битов 5...10 число необходимо умножить на 2^5=32.

Рассмотрим еще пример 48.2 – переименование имеющегося файла, для чего предусмотрена функция 56h. Она требует указания двух спецификаций файла – исходной, чтобы можно было его найти, и новой, присваиваемой файлу. Если в обеих спецификациях указан один и тот же путь к файлу (или имеется только имя файла), осуществляется его переименование; если же пути различаются, функция выполняет перенос файла в другой каталог. Полезно понимать, что перенос файла в другой каталог не требует копирования файла на другое место диска; файл остается на том же месте, изменяются только записи об этом файле в каталогах.

Адрес исходной спецификации должен содержаться в регистрах DS:DX; адрес новой спецификации – в регистрах ES:DI. Это требует при использовании данной функции настройки на сегмент данных программы не только регистра DS, как обычно, но и регистра ES.

Пример 48.2. Переименование файла

```
;В сегменте команд
        push   DS                  ;Настроим ES на
        pop    ES                  ;наш сегмент данных
        mov    AH,56h              ;Функция переименования
        mov    DX,offset fname1    ;Исходная спецификация
        mov    DI,offset fname2    ;Новая спецификация
        int    21h
;В сегменте данных
fname1  db     'myfile.001',0
fname2  db     'myfile01.dat',0
```

Статья 49. **Поиск файлов**

Иногда возникает необходимость найти в некотором каталоге все файлы, удовлетворяющие условиям шаблона групповой операции (например, все файлы с расширением .ASM или все файлы с именем EXAMPLE и любыми расширениями). Поиск файлов по заданным шаблонам групповых операций осуществляется с помощью функций 4Eh (найти первый файл) и 4Fh (найти следующий файл). Для их использования необходимо с помощью функции 1Ah организовать в программе область дисковой передачи данных (Disk transfer area, DTA) размером не менее 43 байт либо с помощью функции 2Fh получить адрес области передачи данных, созданной DOS. Известно, впрочем, что в качестве DTA DOS использует область PSP от байта 80h. DOS помещает в DTA информацию о найденном файле (атрибуты, время и дата создания, размер и т. д.). Содержимое области дисковой передачи данных приведено в табл. 49.1.

Таблица 49.1. Формат DTA

Смещение	*Число байтов*	*Описание*
00h	1	Буква дисковода
01h	11	Шаблон для поиска
0Ch	1	Атрибуты поиска

0Dh	2	Счетчик элементов внутри каталога
0Fh	2	Номер первого кластера родительского каталога
11h	4	Зарезервировано
15h	1	Атрибуты найденного файла
16h	2	Время создания найденного файла
18h	2	Дата создания найденного файла
1Ah	4	Размер найденного файла
1Eh	13	Имя и расширение файла в виде строки ASCIIZ

При поиске файлов по заданному шаблону сначала активизируется функция 4Eh. В регистры DS:DX помещается адрес строки ASCIIZ, в которой задан путь к рассматриваемому каталогу, а в регистр CX – код комбинации атрибутов искомого файла. Если установлены атрибуты поиска, то ищутся как нормальные файлы, так и файлы с заданными атрибутами. В случае успешного нахождения заданного файла функция возвращает CF=0, а имя и расширение файла в виде строки ASCIIZ помещаются в DTA, в байты 1Eh...2Ah. Получив имя файла, можно открыть этот файл с помощью функции 3Dh и выполнить далее требуемые операции (чтение, запись и т. д.).

Поиск следующих файлов, удовлетворяющих условиям заданного шаблона, осуществляется с помощью функции 4Fh, которая используется так же, как и функция 4Eh поиска первого файла. При необходимости функцию 4Fh можно активизировать многократно, пока CF=1 не покажет, что все файлы, удовлетворяющие условиям заданного шаблона, исчерпаны.

Рассмотрим в качестве примера задачу определения метки тома установленной на дисководе дискеты и сравнение ее с требуемой меткой (пример 49.1). Такая операция может использоваться в тех случаях, когда программа обработки данных должна считывать исходные данные с нескольких дискет в установленном порядке. Проверка меток тома позволит обнаружить факт случайной установки на дисковод дискет в ошибочном порядке.

Известно, что метка тома хранится на дискете (как и на жестком диске) в виде записи в корневом каталоге, совпадающей по формату с записью о файле. Единственным отличием является атрибут, который для метки тома должен иметь значение 8 (см. табл. 29.2). Таким образом, для поиска метки следует использовать функцию 4Eh, а поиск выполнять лишь в корневом каталоге дискеты.

Пример 49.1. Проверка метки тома дискеты

```
;В сегменте команд
;Установим нашу область передачи диска (DTA),
;чтобы не искать ее в PSP
        mov    AH,1Ah              ;Функция установки DTA
        mov    DX,offset dta       ;Адрес нашей DTA
        int    21h
;Найдем метку
        mov    AH,4Eh              ;Функция поиска первого файла
```

```
        mov     CX,8                  ;Атрибут "метка тома"
        mov     DX,offset dname       ;Адрес спецификации файлов
        int     21h
        jc      nolabel               ;Метка отсутствует
;Сравним метки
        mov     SI,offset lbl         ;Адрес имени требуемой метки
        mov     AX,seg dta            ;Настроим сегментный
        mov     ES,AX                 ;регистр ES на наш сегмент
        mov     DI,offset dta+1Eh     ;Место для имени файла в DTA
        mov     CX,lbllen             ;Длина имени метки
        cld                           ;Будем сравнивать вперед
repe    cmpsb                         ;Сравнение
        jne     error                 ;Метки не совпадают
;Выведем сообщение mes1 о том, что на дисководе стоит требуемая дискета
        mov     DX,offset mes1
        jmp     go_on
nolabel:
;Выведем сообщение mes2 об отсутствии метки на дискете
        mov     DX,offset mes2
        jmp     go_on
error:
;Выведем сообщение mes3 о том, что на дисководе не та дискета
        mov     DX,offset mes3
go_on:  mov     AH,09h
        int     21h
;В сегменте данных
dname   db      'A:\*.*',0            ;Просмотрим все имена в корневом каталоге
A:\
dta     db      43 dup(0)             ;Область передачи данных
lbl     db      'DATADISK.005'
lbllen=$-lbl
mes1    db      'На дисководе стоит правильная дискета$'
mes2    db      'На дискете нет метки!$'
mes3    db      'На дисководе не та дискета!$'
```

Разумеется, в реальной программе должен быть предусмотрен переход на продолжение программы в случае установки правильной дискеты.

Статья 50. **Ввод с клавиатуры**

Операционная система предоставляет несколько способов ввода данных с клавиатуры:

- обращение к клавиатуре с помощью файловой функции 3Fh прерывания 21h;
- использование группы функций ввода-вывода из диапазона 01h...0Ch прерывания 21h;
- посимвольный ввод путем обращения непосредственно к BIOS с помощью прерывания INT 16h.

Ввод с клавиатуры средствами файловой системы (INT 21h, функция 3Fh) осуществляется точно так же, как и чтение из файла. Обычно используется предопределенный дескриптор 0, закрепленный за стандартным устройством ввода, т. е. за клавиатурой. Число вводимых символов указывается в регистре CX, однако ввод завершается лишь после того, как нажата клавиша Enter, независимо от того, введено ли фактически меньше символов, чем было запланировано, или больше (последнее, естественно, может случиться лишь при неправильных действиях). Поэтому при вводе строк с клавиатуры нет необходимости заранее задавать их длину, достаточно загрузить в регистр CX максимальную длину строки, например 80 байт. В любом

случае в регистре AX возвращается число реально введенных байтов, при этом учитываются также и 2 байта (13 и 10), поступающие во входной буфер при нажатии клавиши Enter.

Особая ситуация возникает, если попытаться ввести больше символов, чем затребовано функцией 3Fh. В процессе выполнения этой функции все вводимые символы тут же извлекаются из кольцевого буфера ввода и пересылаются в буфер DOS. Обнаружив во входном потоке коды клавиши Enter, DOS пересылает из этого буфера в буфер пользователя в программе точно затребованное число символов (естественно, без кодов Enter, которые располагаются в конце введенной строки). Остальные символы остаются в буфере DOS, готовые к вводу. Если не принять специальных мер к очистке буфера, они поступят в программу при очередном запросе 3Fh, даже если оператор еще не начал вводить очередную порцию данных. Очевидно, что в этом случае будет нарушена синхронизация хода выполнения программы с работой оператора.

Второй способ получения данных с клавиатуры в программу, с помощью функций DOS из диапазона 01h...0Ch, несколько более громоздок, но обеспечивает более разнообразные возможности. Вообще отличие этих функции DOS, как уже отмечалось ранее (см. статью 44), заключается в том, что они выполняются на отдельном стеке DOS и, следовательно, к ним можно обращаться в обработчике аппаратных прерываний, если в основной программе используются лишь функции "дисковой" группы (т. е., попросту, все остальные функции DOS). Однако в программах, не связанных с обработкой аппаратных прерываний, принадлежность функции к той или иной группе не имеет особого значения и при выборе для использования в программе следует руководствоваться чисто функциональными соображениями.

Функции ввода с кратким описанием перечислены в табл. 50.1.

Таблица 50.1. Функции ввода из группы функций ввода-вывода

Функция	*Назначение*
01h	Ввод символа с эхом и отработкой Ctrl+C
06h	Прямой посимвольный ввод-вывод через консоль
07h	Ввод символа без эха и без отработки Ctrl+C
08h	Ввод символа без эха с отработкой Ctrl+C
0Ah	Буферизованный ввод строки с эхом
0Bh	Проверка состояния стандартного устройства ввода
0Ch	Сброс кольцевого буфера клавиатуры и ввод одной из функций

Функции 01h, 06h, 07h и 08h при каждом вызове вводят в программу один символ из кольцевого буфера ввода; при необходимости ввести группу символов (строку) эти функции следует использовать в цикле. Различаются они наличием или отсутствием отображения вводимого символа на экране (эха), а также реакцией на ввод с клавиатуры сочетания Ctrl+C. Функции 01h и 0Ah отображают вводимые символы на экране; функции 07h и 08h этого не делают, что дает возможность вводить данные тайком от окружающих (например, пароль или ключ). Второе различие описываемых функций касается их реакции на ввод сочетания Ctrl+C. При выполнении функций 01h и 08h DOS проверяет каждый введенный символ и, обнаружив во входном потоке код Ctrl+C (03h), аварийно завершает программу. Функции же 06h и 07h пропускают код Ctrl+C в программу, не инициируя по нему никаких специальных действий. Таким образом, если программа осуществляет ввод с клавиатуры функциями 06h или 07h, ее нельзя ава-

рийно завершить командой Ctrl+C. В настоящее время, когда известна и широко используется методика перехвата прерывания 23h с передачей управления на прикладной обработчик команд Ctrl+C и Ctrl+Break, все эти рассуждения о нечувствительности к Ctrl+C потеряли свое значение.

Функции 01h, 07h и 08h предназначены только для вывода; функция 06h реализует как посимвольный ввод с клавиатуры, так и вывод символов на экран. Режим работы этой функции задается в регистре AL: код FFh обозначает ввод, любой другой код приводит к выводу на экран соответствующего этому коду символа.

Функция 0Ah передает в буфер пользователя строку, введенную с клавиатуры. Строка должна заканчиваться нажатием клавиши Enter, и ее длина не должна превышать 254 символов. Вводимые символы отображаются на экране; при вводе Ctrl+C происходит аварийное завершение программы.

Функция 0Bh позволяет проверить наличие в кольцевом буфере ввода ожидающих символов. При обнаружении символов программа должна извлечь их из буфера одной из функций ввода; если символов нет, программа может продолжить выполнение. Такая методика используется в программах, носящих циклический характер, если требуется обеспечить управление ходом выполнения программы с клавиатуры терминала. В каждом шаге цикла после выполнения запланированных действий проверяется состояние кольцевого буфера ввода; если в течение предыдущего шага цикла оператор нажал на какую-либо клавишу, программа проанализирует введенный код и осуществит выход из цикла и переход в ту или иную точку; если же буфер оказывается пуст, циклическое выполнение продолжится.

Функция 0Bh чувствительна к Ctrl+C. Это дает возможность организовать с ее помощью аварийное завершение программы на тех ее участках, где выполняются чисто процессорные действия. Если, например, включить вызов функции 0Bh в цикл, то при отсутствии ввода с клавиатуры цикл будет выполняться обычным образом, но после ввода Ctrl+C программа аварийно завершится, хотя на выполняемом участке программы не используются функции ввода-вывода.

Функция 0Ch служит для организации ввода с предварительной очисткой кольцевого буфера. Все функции, кроме 0Ch, вводят в программу наиболее старый из скопившихся в кольцевом буфере ввода символов, реализуя тем самым возможность ввода с упреждением. В этом режиме оператор может нажимать на клавиши еще до выдачи программой запроса на ввод; коды нажатых клавиш (не более 15) будут накапливаться в кольцевом буфере ввода и извлекаться оттуда в программу по мере выполнения ею запросов на ввод. В отличие от этого функция 0Ch сначала очищает кольцевой буфер и лишь затем ожидает ввода символа с клавиатуры. В результате коды всех ранее нажатых (по предположению – случайно) клавиш теряются. Обычно функцию 0Ch используют в программе непосредственно вслед за функцией вывода на экран символьной строки с предложением оператору вводить данные. В результате из кольцевого буфера убирается весь "мусор" от случайных нажатий, в программу же поступает лишь то, что вводится оператором после запроса программы. При этом режим ввода (с эхом или без него и т. д.) определяется тем, какая именно функция ввода (01h, 07h, 08h или 0Ah) реализуется "внутри" функции 0Ch.

Функции 01h, 07h, 08h и 0Ah являются синхронными, т. е. при отсутствии символа в кольцевом буфере ждут его ввода. Функция 06h позволяет определить состояние

кольцевого буфера и при наличии в нем кода извлечь этот код и обработать его, а при отсутствии – продолжить выполнение программы.

Функции 01h, 06h, 07h и 08h позволяют вводить в программу расширенные коды ASCII. Для этого, обнаружив, что введенный код ASCII равен нулю, следует выполнить функцию повторно. Это дает возможность управления прикладными программами с помощью функциональных клавиш, а также сочетаний Alt+цифра, Alt+буква и др.

Функции 06h, 07h и 08h позволяют вводить в программу коды символов с помощью сочетания Alt+цифра на цифровой клавиатуре (в том числе некоторые из первых 32 символов кодовой таблицы и всю вторую половину кодовой таблицы).

Работа с клавиатурой на уровне BIOS (INT 16h) позволяет считывать 2-байтовые коды, поступающие в кольцевой буфер ввода (код ASCII + скан-код) и анализировать слово флагов клавиатуры (нажатие клавиш Ctrl, Alt, Shift и др.). Функции BIOS, используемые для ввода с клавиатуры, перечислены в табл. 50.2.

Таблица 50.2. Функции ввода BIOS (прерывание int 16h)

Функция	*Назначение*
00h	Чтение 2-байтового кода из входного буфера
01h	Чтение состояния клавиатуры и 2-байтового кода без извлечения его из буфера
02h	Чтение флагов клавиатуры

Функция 00h позволяет в одном действии получить полный 2-байтовый код нажатой клавиши или комбинации клавиш, из которого, в частности, можно извлечь скан-код (некоторые программы идентифицируют нажатые клавиши не по кодам ASCII, а по их скан-кодам), а также получить значащую часть расширенного кода ASCII (при нажатии, например, функциональных клавиш). Функция 00h является синхронной: при ее выполнении программа останавливается в ожидании нажатия клавиши.

Функция 01h относится к числу асинхронных: определив состояние клавиатуры (точнее, буфера ввода), она возвращает управление программе. Состояние буфера возвращается во флаге ZF: если в буфере имеются ожидающие ввода в программу символы, ZF=0; если же буфер пуст, ZF=1. При наличии в буфере кода символа его можно проанализировать, так как он возвращается функцией в регистре AX (AH=скан-код, AL=код ASCII). Необходимо, однако, иметь в виду, что функция 01h, копируя 2-байтовый код в регистр AX, не очищает при этом кольцевой буфер. Забрать символ с очисткой буфера можно затем функцией 00h.

Функция 02h передает в программу содержимое слова флагов (ячейка 417h). Она может использоваться программами, работающими на уровне скан-кодов, для определения состояния клавиш Shift, Caps Lock и др.

Рассмотрим несколько примеров использования системных функций для ввода данных с клавиатуры.

Пример 50.1. Ввод с клавиатуры строки с помощью файловой функции 3Fh

```
;В сегменте команд
;Выведем на экран строку-запрос
        mov    AH,09h
        mov    DX,offset msg
        int    21h
;Поставим запрос на ввод
        mov    AH,3Fh        ;Файловая функция ввода
        mov    BX,0          ;Дескриптор клавиатуры
```

```
        mov   CX,25        ;Максимальное число символов
        mov   DX,offset inbuf;Адрес программного буфера ввода
        int   21h
        ...
;В сегменте данных
msg     db    'Введите фамилию: $'
inbuf   db    256 dup ('*'),'$'
```

В примере 50.1 на экран выводится строка-запрос пользователю, после чего функция 3Fh ожидает ввода строки, завершающейся нажатием клавиши Enter. По мере ввода (до нажатия клавиши Enter) строку можно редактировать: стирать введенные символы и заменять их новыми. После нажатия клавиши Enter введенная строка вместе с кодами 13 и 10 поступает в программный буфер inbuf, где ее можно обнаружить с помощью отладчика.

Пример 50.2. Управление программой с помощью функциональных клавиш

```
;В сегменте команд
;Выведем на экран строку-запрос
        mov   AH,09h
        mov   DX,offset reqst
        int   21h
;Фильтрация расширенных кодов ASCII
        mov   AH,08h       ;Функция ввода символа без эха
again:  int   21h
        cmp   AL,0         ;Младший байт = 0?
        jne   again        ;Нет, повторять ввод символа
        mov   AH,08h       ;Да, введем старший байт
        int   21h
        cmp   AL,3Bh       ;Нажата клавиша F1?
        je    f1           ;Да, на соответствующий фрагмент
        cmp   AL,3Ch       ;Нажата клавиша F2?
        je    f2           ;Да, на соответствующий фрагмент
        cmp   AL,3Dh       ;Нажата клавиша F3?
        je    f3           ;Да, на соответствующий фрагмент
        jmp   again        ;Нажато незапланированное
f1:     ;Фрагмент, выполняемый по команде F1
        mov   AH,09h
        mov   DX,offset msg_f1
        int   21h
        jmp   go           ;На продолжение программы
f2:     ;Фрагмент, выполняемый по команде F2
        mov   AH,09h
        mov   DX,offset msg_f2
        int   21h
        jmp   go           ;На продолжение программы
f3:     ;Фрагмент, выполняемый по команде F3
        mov   AH,09h
        mov   DX,offset msg_f3
        int   21h
go:     ;Продолжение программы (у нас - завершение)
;В сегменте данных
reqst   db    'Вводите команду (F1, F2 или F3):',13,10,'$'
msg_f1  db    'Введена команда F1$'
msg_f2  db    'Введена команда F2$'
msg_f3  db    'Введена команда F3$'
```

Как уже было отмечено, для обнаружения во входном потоке расширенных кодов ASCII необходимо анализировать поступающие коды и при обнаружении кода 0 повторить ввод символа. Этот повторный ввод приведет к извлечению из кольцевого бу-

фера. старшего, значащего байта 2-байтового расширенного кода ASCII. В примере 50.2 ввод осуществляется функцией 08h без эха, что избавляет нас от появления на экране бессмысленных символов. Полученный старший байт кода последовательно проверяется на совпадение с кодами ASCII клавиш F1, F2 и F3, после чего выполняется переход на соответствующий фрагмент программы.

Пример 50.3. Ввод с клавиатуры с очисткой кольцевого буфера

```
;В сегменте команд
;Выведем на экран информационную строку
        mov   AH,09h
        mov   DX,offset msg
        int   21h
;Введем в программу задержку порядка нескольких секунд
        mov   CX,5000
delay2: push  CX
        mov   CX,0
delay1: loop  delay1
        pop   CX
        loop  delay2
;Выведем на экран строку-запрос
        mov   AH,09h
        mov   DX,offset reqst
        int   21h
;Поставим запрос на ввод символа с очисткой кольцевого буфера
again:  mov   AH,0Ch
        mov   AL,01h
        int   21h
        cmp   AL,'q'
        jne   again
        ...                 ;Завершим программу
;В сегменте данных
msg     db    'Программа стартовала',13,10,'$'
reqst   db    'Вводите команду: $'
```

В примере 50.3 с помощью функции 01h осуществляется циклический посимвольный ввод с клавиатуры с отображением вводимых символов на экране. Нажатие клавиши Q (на нижнем регистре) приводит к завершению программы. Для наглядной демонстрации действия функции 0Ch ввода с очисткой кольцевого буфера в начале программы на экран выводится информационное сообщение, после чего выполняется программная задержка достаточной длительности. В течение этой задержки пользователь может нажимать на любые клавиши, в том числе и (ошибочно) на "завершающую" клавишу Q. Весь этот "мусор" удаляется из кольцевого буфера функцией 0Ch, после чего активизируется функция ввода 01h, останавливающая программу в ожидании ввода с клавиатуры символа-команды. При поступлении символа 'q' программа завершается.

Дополнительные примеры использования функций ввода с клавиатуры в прикладных программах будут приведены в статье 52.

Статья 51. **Вывод на экран средствами DOS**

Как и при вводе с клавиатуры, операционная система предоставляет несколько способов вывода данных на экран:

- обращение к экрану с помощью файловой функции 40h прерывания 21h;

- использование группы функций ввода-вывода из диапазона 01h...0Ch прерывания 21h;
- вывод на экран средствами BIOS с помощью прерывания INT 10h.

DOS (прерывание 21h) поддерживает только текстовый монохромный вывод; с помощью функций BIOS можно реализовать все возможности видеосистемы: вывод цветных символов, установку видеорежимов, переключение видеостраниц, загрузку шрифтов и т. д. В настоящей статье будут кратко рассмотрены средства DOS; гораздо более многообразным средствам BIOS будет посвящена следующая статья.

Вывод на экран средствами файловой системы (прерывание 21h, функция 40h) осуществляется точно так же, как и запись в файл (пример 51.1). Используются предопределенные дескрипторы 1 (стандартный вывод) или 2 (стандартная ошибка), закрепленные за экраном. Вывод через дескриптор 1 может быть перенаправлен в файл или на другое устройство (последовательный порт, принтер); вывод через дескриптор 2 не перенаправляется и обычно используется для вывода на экран служебных сообщений. Число выводимых символов указывается в регистре CX, а адрес выводимой строки – в регистрах DS:DX. Коды 08h (возврат на шаг), 0Ah (перевод строки), 0Dh (возврат каретки) и некоторые другие рассматриваются как управляющие и приводят к выполнению соответствующих им действий.

Пример 51.1. Вывод на экран средствами файловой системы

```
;В сегменте команд
        mov     AH,40h          ;Функция вывода
        mov     BX,1            ;Дескриптор экрана
        mov     CX,strlen       ;Длина строки
        mov     DX,offset msg   ;Адрес строки
        int     21h             ;Вызов DOS
;В сегменте данных
msg     db 'Программа начинает свою работу',13,10
msglen=$-msg                    ;Длина строки
```

Как уже отмечалось, DOS не поддерживает цвет и, кроме того, не предоставляет средств очистки экрана и позиционирования курсора. Таким образом, вывод с помощью функций DOS осуществляется только строка за строкой, причем по мере вывода строк изображение на экране автоматически прокручивается вверх. Однако в состав DOS включается драйвер ANSI.SYS, установка которого (с помощью файла CONFIG.SYS) несколько расширяет возможности вывода на экран. В этом случае на экран, помимо содержательных текстов, посылаются Esc-последовательности, которые позволяют изменять цвет символов, позиционировать курсор и выполнять некоторые другие настройки экрана (и клавиатуры). Использование Esc-последовательностей было описано в статье 12.

Второй способ вывода на экран текстовой информации реализуется с помощью трех функций DOS, приведенных в табл. 51.1.

Таблица 51.1. Функции DOS вывода на экран

Функция	*Назначение*
02h	Вывод символа
06h	Прямой ввод-вывод
09h	Вывод строки

Функция 09h (пример 51.2) широко используется в системных и прикладных программах для вывода на экран информационных и диагностических сообщений. Она не требует указания дескриптора (хотя в действительности использует дескриптор стандартного вывода 1); нет необходимости также указывать длину выводимой строки. Перед вызовом прерывания адрес сообщения заносится в регистры DS:DX; сообщение должно заканчиваться символом $ (код которого равен 24h). При отсутствии в строке этого символа на экран будет выводиться содержимое всех байтов памяти, расположенных за строкой, до тех пор, пока в памяти случайно не встретится код 24h.

Пример 51.2. Вывод на экран строки с помощью функции 09h

```
;В сегменте команд
        mov    AH,09h           ;Функция вывода
        mov    DX,offset msg    ;Адрес строки
        int    21h              ;Вызов DOS
;В сегменте данных
msg     db 'Драйвер установлен$',13,10
msglen=$-msg                    ;Длина строки
```

Функция 02h (пример 51.3) вызывает передачу на экран одного символа, помещаемого в регистр DL. Для вывода последовательности символов функцию нужно использовать в цикле.

Пример 51.3. Вывод на экран символа с помощью функции 02h.

```
        mov    AH,02h           ;Функция вывода
        mov    DL,'?'           ;Выводимый символ
        int    21h              ;Вызов DOS
```

Функция 06h не имеет особых преимуществ перед другими функциями вывода и используется реже.

Статья 52. **Вывод на экран средствами BIOS**

Функции BIOS обладают большими возможностями и широко используются в прикладном программировании для формирования цветных информационных кадров, переключения видеорежимов, загрузки шрифтов пользователя и других действий с видеосистемой. Их недостатком в сравнении с функциями DOS является относительная громоздкость использования. Функции, чаще других используемые при работе в текстовом режиме, перечислены в табл. 52.1.

Таблица 52.1. Основные функции BIOS для работы в текстовом режиме

Функция	*Назначение*
02h	Установить позицию курсора
03h	Получить позицию курсора
05h	Установить видеостраницу
06h	Инициализировать или прокрутить вверх окно
07h	Инициализировать или прокрутить вниз окно
08h	Прочитать символ и атрибут в позиции курсора
09h	Вывести символ и атрибут в позицию курсора
0Ah	Вывести символ в позицию курсора
0Eh	Вывести символ в режиме телетайпа
1003h	Переключить бит мерцание/яркость
13h	Вывести строку в режиме телетайпа

Функция 02h позволяет позиционировать текстовый курсор, задавая его местоположение в виде номера строки (0...24) и номера столбца (0...79). BIOS поддерживает 8 независимых курсоров – по одному на каждую видеостраницу, причем функция 02h позиционирует курсор независимо от того, какая страница является активной.

Функция 03h позволяет получить и сохранить текущее положение курсора. Это дает возможность перейти временно в другое место экрана, сформировать там изображение, а затем вернуться на старое место.

Функция 05h переключает видеостраницы. Если видеосистема находится в текстовом режиме, то переключаются текстовые страницы (0...7), если установлен графический режим, то переключаются графические страницы (0...1).

Большая часть описываемых ниже функций вывода на экран (кроме подфункции переключения мерцания/яркости и функции вывода строки в режиме телетайпа) позволяют формировать изображение на любой видеостранице, как активной в настоящей момент, так и скрытой. Это дает возможность либо подготовить заранее несколько страниц и по мере необходимости быстро их переключать, либо, пока одна страница выводится на экран, готовить изображение на следующей.

С помощью функций 06h и 07h в заданном месте экрана дисплея создаются цветные прямоугольные окна заданного размера. Если в созданные ранее окна выведен какой-либо текст, то с помощью этих же функций можно прокручивать текст вверх или вниз. При этом текст, уходящий за край окна, пропадает, а из-под противоположного края появляются пустые строки с заданными атрибутами цвета. Для заполнения появляющихся строк текстом следует использовать подходящие функции BIOS, причем контроль местоположения, длины и цвета строк возлагается на программиста. Программы BIOS только прокручивают заданную прямоугольную область экрана (вместе с текстом в ней).

Функции 09h, 0Ah, 0Eh и 13h служат для вывода на экран отдельных символов и символьных строк (в цикле). Функции 09h и 0Ah не выполняют фильтрации управляющих символов, поэтому с их помощью можно выводить все символы кодовой страницы. Предусмотрен вывод одного и того же символа заданное число раз, что можно использовать при создании рамок и орнаментов. Вывод символа не перемещает курсор, поэтому каждый раз перед применением функций 09h или 0Ah следует позиционировать курсор с помощью функции 02h. Различие функций 09h и 0Ah заключается в том, что первая позволяет вывести символ с любым атрибутом, а вторая использует прежний атрибут той позиции, куда выводится символ.

Функция 0Eh фильтрует управляющие коды 07h (звуковой сигнал), 08h (возврат на шаг), 10h (перевод строки) и 13h (возврат каретки), выполняя соответствующие им действия. Курсор перемещается после вывода каждого символа, что дает возможность выводить целые строки. Однако атрибут символа установить нельзя, выводимый символ приобретает прежний атрибут той позиции, куда он выводится. При необходимости вывода символа с новым атрибутом следует сначала вывести в заданную позицию символ пробела с требуемым атрибутом (функцией 09h), а затем туда же послать символ с помощью функции 0Eh.

Важным свойством функции 0Eh является автоматический переход на следующую строку после завершения предыдущей, а также прокрутка экрана вверх на одну строку после заполнения самой нижней строки.

Функция 13h предназначена для вывода строк с указанием атрибутов как каждого символа в отдельности, так и всей строки. Функция может выполняться в четырех вариантах в зависимости от кода режима, указываемого в регистре AL. В режимах 0 и 1 атрибут символов указывается сразу для всей строки в регистре BL, причем в режиме 0 курсор не смещается в процессе вывода, а в режиме 1 – смещается на длину строки. В режимах 2 и 3 атрибуты символов включаются в выводимую строку, в которой, таким образом, чередуются коды атрибутов и коды символов, что усложняет формат строки, но позволяет устанавливать атрибуты для каждого символа независимо. Режим 2 отличается от режима 3 тем, что в первом случае курсор не смещается, а во втором смещается на длину строки.

При вызове функции 13h в регистре DX задаются координаты начала выводимой строки (в DH – строка экрана, и в DL – столбец), а в регистре CX – длина выводимой строки, которая в режимах 2 и 3 оказывается за счет байтов с атрибутом в два раза больше длины строки, реально появляющейся на экране. Адрес выводимой строки должен быть помещен в регистры ES:BP.

Функция 13h выводит не все символы, так как коды 07h, 08h, 0Ah и 0Dh рассматриваются ею как управляющие.

При выводе на экран средствами BIOS необходимо иметь в виду, что ввод с клавиатуры Ctrl+C не приводит к завершению программы. Следует опасаться бесконечных циклов вывода на экран; выход из них возможен только путем перезагрузки компьютера.

Подфункция 03h функции 10h (прерывание 10h), в отличие от описанных выше функций вывода символов и строк, воздействует сразу на весь экран, влияя на отображение тех символов, у которых установлен старший бит атрибута фона. Функция позволяет либо приписать этот бит яркости фона, давая тем самым возможность выводить на экран 16 цветов фона, либо назначить его атрибуту мерцания символа. В последнем случае цвет фона может принимать только 8 значений.

Рассмотрим несколько примеров использования функций прерывания BIOS 10h для вывода информации на экран.

Пример 52.1. Вывод на экран цветных символов средствами BIOS

```
;В сегменте команд
        mov  AX,data
        mov  DS,AX
;Выполним начальную настройку регистров
        mov  CX,7           ;Число выводимых символов
        mov  DL,36          ;Начальная позиция на строке экрана
        mov  SI,offset msg;Смещение в строке текста
output:
;Позиционируем курсор
        mov  AH,02h         ;Функция установки курсора
        mov  BH,0           ;Видеостраница
        mov  DH,12          ;Строка
        int  10h            ;Прерывание BIOS
;Выводим символ
        mov  AH,09h         ;Функция вывода символа
        mov  AL,[SI]        ;Символ
        mov  BL,34h         ;Атрибут
        push CX             ;Сохраним на время CX
        mov  CX,1           ;Коэффициент повторения
        int  10h            ;Прерывание BIOS
        pop  CX             ;Восстановим CX
```

```
        inc  SI             ;Сдвиг по строке текста
        inc  DL             ;Сдвиг по экрану
        loop output         ;Цикл
;В сегменте данных
msg     db   'Авария!'
```

В примере 52.1 в центр экрана выводится фраза 'Авария!' красными буквами по бирюзовому фону. Мы видим, что вывод каждого отдельного символа выполняется с помощью двух программных блоков: установки позиции курсора и собственно вывода символа. Поскольку функция 09h не перемещает курсора, это приходится делать вручную перед выводом каждого следующего символа. С другой стороны, используя функции BIOS, мы имеем возможность выводить цветные строки в любое место экрана.

Пример 52.2. Организация окна и вывод символьной таблицы

```
;Очистим экран и зададим атрибуты символов с помощью окна
        mov   AH,06h        ;Функция инициализации окна
        mov   AL,0          ;Не прокручивать
        mov   BH,31h        ;Бирюзовый фон,синие символы
        mov   CH,0          ;Левая верхняя координата y
        mov   CL,0          ;Левая верхняя координата x
        mov   DH,24         ;Правая нижняя координата y
        mov   DL,79         ;Правая нижняя координата x
        int   10h
;Установим начальные значения в регистрах
        mov   DH,10         ;Начальная строка
        mov   DL,10         ;Начальный столбец
        mov   AL,0          ;Начальный символ
;Будем выводить символы четырьмя строками по 64 символа
        mov   CX,4          ;4 строки (внешний цикл)
rows:   push  CX            ;Сохраним счетчик внешнего цикла
        mov   CX,64         ;64 столбца (внутренний цикл)
colms:  push  CX            ;Сохраним счетчик внутреннего цикла
;Позиционируем курсор
        mov   AH,02h        ;Функция позиционирования
        mov   BH,0          ;Страница 0
        int   10h
;Выведем очередной символ
        mov   AH,0Ah        ;Функция вывода символа без
                            ;задания его атрибута
        mov   CX,1          ;Коэффициент повторения
        int   10h
;Установим новые значения переменных и организуем циклы
        inc   AL;char       ;Следующий символ
        inc   DL            ;Следующий столбец
        pop   CX ;Восстановим счетчик внутреннего цикла
        loop  colms         ;Цикл по столбцам
        mov   DL,10         ;Снова с начала строки
        add   DH,2          ;Вниз на две строки
        pop   CX ;Восстановим счетчик внешнего цикла
        loop  rows          ;Цикл по строкам
```

Функция 06h, с помощью которой создается цветное окно размером во весь экран, используется в данном примере для очистки экрана и заодно для задания всем знакоместам экрана одинаковых атрибутов. После этого вывод символов удобно выполнять с помощью функции 0Ah, которая не требует задания атрибута. Как и в предыдущем примере, перед выводом очередного символа приходится с помощью функции 02h устанавливать позицию курсора. Центральная область экрана с выводом программы 52.2 показана на рис. 52.1.

Как уже отмечалось выше (см. статью 24), видеосистема компьютера предоставляет возможность использовать в обычном текстовом режиме (80x25 символов) 8 видеостраниц, каждая из которых может независимо заполняться требуемым содержимым. DOS работает только с текущей видеостраницей; для заполнения других видеостраниц, а также для переключения текущей видеостраницы следует использовать функции прерывания BIOS 10h. Методика переключения видеостраниц используется, например, в отладчиках, которые на одной видеостранице хранят свой информационный кадр с текстом отлаживаемой программы и инструментами отладки, а другую видеостраницу используют для показа "экрана DOS", т. е. выводимых на экран результатов работы отлаживаемой программы.

```
☺☻♥♦♣♠•◘○◙♂♀♪♫☼►◄↕‼¶§▬↨↑↓→←∟↔▲▼ !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~⌂
АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмноп░▒▓│┤╡╢╖╕╣║╗╝╜╛┐
└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌█▄▌▐▀рстуфхцчшщъыьэюяЁёЄєЇїЎў°∙·√№¤■
```

Рис. 52.1. Полная таблица символов, выведенная на экран средствами BIOS

В примере 52.3 продемонстрирована методика использования двух видеостраниц для переключения, по команде пользователя, информационных кадров с разнородной информацией. На страницу 0 программа выводит текущие результаты своей работы; при нажатии любой клавиши осуществляется переключение на страницу 1, которая показывает, например, текущее время или какую-либо другую динамически изменяемую информацию. Повторное нажатие любой клавиши переключает программу на продолжение работы на странице 0. В данном примере на страницу 0 выводятся последовательные символы второй половины кодовой таблицы. Вывод осуществляется в цикле с небольшой задержкой, чтобы уменьшить неприятное мерцание экрана. Вторая половина таблицы выбрана потому, что все ее символы могут отображаться на экране, в то время как часть кодов первой половины таблицы, например коды 10 или 13, при использовании для вывода функций DOS (в примере символы выводятся функцией DOS 02h) не отображаются на экране в виде символов, а выполняют перемещение курсора.

Пример 52.3. Работа с видеостраницами

```
;В сегменте команд
;Настроим регистры DS и ES на сегмент данных
        mov    AX,data      ;(1)
        mov    DS,AX        ;(2)
        mov    ES,AX        ;(3)
;Работаем на странице 0
;Пусть программа в цикле выводит что-то на экран
pg0:    mov    CX,0         ;(4)Небольшая
delay:  loop   delay        ;(5)задержка
        mov    AH,02h       ;(6)Функция вывода символа
        mov    DL,sym       ;(7)Текущий символ
        or     DL,80h       ;(8)Выводим 2-ю половину таблицы
        int    21h          ;(9)Вызов DOS
        inc    sym          ;(10)Инкремент выводимого символа
        mov    AH,0Bh       ;(11)Возможность завершения
        int    21h          ;(12)по Ctrl+C
        mov    AH,06h       ;(13)Нажата ли клавиша?
```

```
        mov   DL,0FFh      ;(14)Режим ввода
        int   21h          ;(15)
        jz    pg0          ;(16)Не нажата, продолжить работу
                           ;на странице 0
;Нажата! Перейдем на страницу 1
pg1:    mov   AH,05h       ;(17)Функция переключения страницы
        mov   AL,1         ;(18)Страница 1
        int   10h          ;(19)
;Прочитаем время из КМОП-микросхемы
;Дождемся возможности чтения
        mov   AL,0Ah       ;(20)Регистр A
        out   70h,AL       ;(21)
wait1:  in    AL,71h       ;(22)Проверка на
        test  AL,80h       ;(23)наличие цикла
        jnz   wait1        ;(24)коррекции
;Прочитаем минуты и преобразуем в ASCII
        mov   AL,02h       ;(25)Ячейка минут КМОП-памяти
        out   70h,AL       ;(26)
        in    AL,71h       ;(27)Прочитали минуты
        mov   BL,AL        ;(28)Скопируем в BL
        shr   BL,4         ;(29)Старшую цифру в начало байта
        add   BL,30h       ;(30)Преобразуем в ASCII
        mov   time,BL      ;(31)Поместим в строку для времени
        and   AL,0Fh       ;(32)Выделим младшую цифру времени
        add   AL,30h       ;(33)Преобразуем в ASCII
        mov   time+1,AL    ;(34)Поместим в строку для времени
;Дождемся возможности чтения
        mov   AL,0Ah       ;(35)Регистр A
        out   70h,AL       ;(36)
wait2:  in    AL,71h       ;(37)Проверка на
        test  AL,80h       ;(38)наличие цикла
        jnz   wait2        ;(39)коррекции
;Прочитаем секунды и преобразуем в ASCII
        mov   AL,0h        ;(40)Ячейка секунд КМОП-памяти
        out   70h,AL       ;(41)
        in    AL,71h       ;(42)Прочитали минуты
        mov   BL,AL        ;(43)Скопируем в BL
        shr   BL,4         ;(44)Сдвинем старшую цифру в начало байта
        add   BL,30h       ;(45)Преобразуем в ASCII
        mov   time+7,BL    ;(46)Поместим в строку для времени
        and   AL,0Fh       ;(47)Выделим младшую цифру времени
        add   AL,30h       ;(48)Преобразуем в ASCII
        mov   time+8,AL    ;(49)
;Выведем на странице 1 строку с текущим временем
        mov   AH,13h       ;(50)Функция вывода строки
        mov   AL,0         ;(51)Режим
        mov   BL,04h       ;(52)Атрибут
        mov   BH,1         ;(53)Страница
        mov   CX,timelen   ;(54)Число символов
        mov   DL,30        ;(55)Столбец
        mov   DH,12        ;(56)Строка
        mov   BP,offset time;(57)Выводимая строка
        int   10h          ;(58)Прерывание BIOS
        mov   AH,07h       ;(59)Дождемся нажатия клавиши
        int   21h          ;(60)без анализа Ctrl+C!
;Переключим экран на страницу 0
        mov   AH,05h       ;(61)Функция переключения страницы
        mov   AL,0         ;(62)Страница 0
        int   10h          ;(63)
        jmp   pg0          ;(64)Вернемся к работе на странице 0
;В сегменте данных
```

```
sym      db 0                ;(65)
time     db '** мин ** сек'  ;(66)
timelen=$-time               ;(67)
```

Программа начинается с инициализации сегментных регистров DS и ES (предложения 1...3); через регистр ES осуществляется задание адреса выводимой строки для функции BIOS 13h. Далее начинается работа с видеостраницей 0, устанавливаемой по умолчанию. После небольшой задержки на экран выводится очередной код из ячейки sym (предложения 6...9), содержимое которой увеличивается на единицу в каждом шаге цикла (предложение 10). Чтобы исключить вывод символов первой половины кодовой таблицы (коды 00h...7Fh), в выводимом коде принудительно устанавливается старший бит (предложение 8). Таким образом, на экран последовательно поступают коды 80h...FFh. В цикл вывода включен вызов функции DOS 0Bh (предложения 11 и 12), осуществляющей проверку наличия символа в буфере клавиатуры. Этот вызов обеспечивает возможность завершения программы вводом с клавиатуры комбинаций Ctrl+C или Ctrl+Break.

Для организации бесконечного цикла с возможностью выхода из него по нажатии любой клавиши используется функция DOS 06h (предложения 13...16). При отсутствии символа в буфере клавиатуры эта функция возвращает сброшенным флаг ZF, что приводит к зацикливанию этой части программы (предложение 16). Если же была нажата какая-либо клавиша, осуществляется переход на следующую строку программы с меткой pg1.

После переключения на видеостраницу 1 (предложения 17...19) выполняется чтение из КМОП-микросхемы текущего времени (для сокращения программы только минут и секунд), преобразования получаемых данных из BCD в коды ASCII и сохранение текущего времени в строке time, входящей в сегмент данных. Процедура чтения времени из КМОП-микросхемы рассматривалась в статье 27; преобразование двоично-десятичных кодов в символы – в статье 19.

Сформировав строку time, программа вызывает функцию BIOS 13h, которая позволяет вывести в заданное место экрана текст с указанным атрибутом (предложения 50...58). Адрес выводимой строки для этой функции задается в регистрах ES:BP; регистр ES мы настроили в начале программы, в предложении 57 смещение строки заносится в регистр BP. В результате выполнения функции 13h в центр экрана выводится красным цветом текущее время в следующем виде:

03 мин 42 сек

Для обеспечения возможности переключения назад на страницу 0 используется функция DOS 07h, которая нечувствительна к Ctrl+C и не отображает на экране введенный символ (предложения 59 и 60). Отсутствие чувствительности к Ctrl+C предотвращает выход из программы при активизированной странице 1, что привело бы к нарушению нормальной работы компьютера.

После нажатия любой клавиши экран переключается на страницу 0 (предложения 61...63) и осуществляется переход на метку pg0 (предложение 64), чем организуется бесконечный цикл выполнения программы до ввода (при нахождении на странице 1) команды Ctrl+C.

На рис. 52.2 приведен вывод программы на страницу 0 (цвета фона и символов инвертированы). Как видно из рисунка, работа программы на странице 0 была завершена вводом Ctrl+C.

```
F:\CURRENT>p
АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмноп░▒▓│┤╡╢╖╕╣║╗╝╜╛┐└┴┬├─┼╞╟╚╔╩╦╠═╬
╧╨╤╥╙╘╒╓╫╪┘┌█▄▌▐▀рстуфхцчшщъыьэюяЁёЄєЇїЎў°∙·√№¤■ АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ
абвгдежзийклмноп░▒▓│┤╡╢╖╕╣║╗╝╜╛┐└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌█▄▌▐▀рстуфхцчшщъыьэюя
ЁёЄєЇїЎў°∙·√№¤■ АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмноп░▒▓│┤╡╢╖╕╣║╗╝╜╛┐
└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌█▄▌▐▀рстуфхцчшщъыьэюяЁёЄєЇїЎў°∙·√№¤■ АБВГДЕЖЗИЙКЛМНОП
РСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмноп░▒▓│┤╡╢╖╕╣║╗╝╜╛┐└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌█▄▌▐▀
рстуфхцчшщъыьэюяЁёЄєЇїЎў°∙·√№¤■ АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмноп
░▒▓│┤╡╢╖╕╣║╗╝╜╛┐└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌█▄▌▐▀рстуфхцчшщъыьэюяЁёЄєЇїЎў°∙·√№¤■
АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмноп░▒▓│┤╡╢╖╕╣║╗^C
```

Рис. 52.2. Вывод программы на страницу 0

Статья 53. **Вывод графических изображений. Современные видеорежимы**

В предыдущих статьях было показано, что для осуществления ввода с клавиатуры и вывода на экран символьной информации приходится прибегать к функциям DOS. При этом выяснилось, что возможности DOS довольно скромны. DOS не поддерживает ни позиционирование курсора, ни смену цвета выводимых символов. В текстовом режиме расширить возможности DOS можно с помощью драйвера ANSI.SYS. С графическими изображениями дело обстоит хуже, так как в DOS нет никаких графических функций. Нет их также и в драйвере ANSI.SYS, за исключением возможности перевода видеоадаптера в графический режим (с помощью Esc-последовательности Esc[=*режим*h). Для того чтобы вывести на экран графическое изображение, необходимо воспользоваться нижним уровнем операционной системы – базовой системой ввода-вывода BIOS). Программы BIOS находятся в ПЗУ BIOS, которым в обязательном порядке комплектуется любой компьютер. В отличие от DOS, ко всем функциям которой мы обращаемся с помощью прерывания 21h, в BIOS за каждым устройством компьютера закреплено свое прерывание. Так, обмен с дисковым накопителем осуществляется с помощью прерывания int 13h, с клавиатурой – int 16h, с видеоадаптером – int 10h. Прерывание int 10h позволяет реализовать такие функции, как смена видеорежима, вывод символьной и текстовой информации, смена шрифтов, настройка цветовой палитры, работа с графическим изображением, и др. Проиллюстрируем работу с этим прерыванием на простом примере – выполним переход в графический режим и выведем простейшее графическое изображение (пример 53.1).

Пример 53.1. Вывод на экран горизонтальной прямой

```
;Установим графический режим EGA
        mov   AH,00h      ;(1)Функция задания режима
        mov   AL,10h      ;(2)Графический режим EGA
        int   10h         ;(3)Вызов BIOS
;Нарисуем прямую линию в цикле по координате x
        mov   SI,150      ;(4)Начальная x-координата
        mov   CX,300      ;(5)Число точек по горизонтали
line:   push  CX          ;(6)Сохраним его в стеке
        mov   AH,0Ch      ;(7)Функция вывода пиксела
        mov   AL,4        ;(8)Цвет красный
        mov   BH,0        ;(9)Видеостраница
        mov   CX,SI       ;(10)X-координата (переменная)
        mov   DX,175      ;(11)Y-координата (константа)
        int   10h         ;(12)Вызов BIOS
        inc   SI          ;(13)Инкремент X-координаты
```

```
        pop    CX              ;(14)Восстановим счетчик шагов
        loop   line            ;(15)Цикл из CX шагов
;Остановим программу для наблюдения результата ее работы
        mov    AH,08h          ;(16)Функция ввода с клавиатуры без эха
        int    21h             ;(17)Вызов DOS
;Переключим видеоадаптер назад в текстовый режим
        mov    AH,00h          ;(18)Функция задания режима
        mov    AL,03h          ;(19)Текстовый режим
        int    10h             ;(20)Вызов BIOS
```

В предложениях 1...3 с помощью функции 00h прерывания BIOS 10h осуществляется переключение видеоадаптера в графический режим. Поскольку номер режима заносится в байтовый регистр AL, всего может существовать 256 различных текстовых и графических режимов, из которых на сегодняшний день используются (аппаратурой различных фирм) около ста. Режим 10h обеспечивает вывод графического изображения 16 цветами с разрешением 640*350 точек и широко используется при работе с видеоадаптерами EGA и VGA.

Изображение рисуется по точкам (в BIOS не предусмотрено программных средств вывода каких-либо геометрических фигур или хотя бы линий, как нет и средств закрашивания областей экрана). Для вывода на экран цветной точки (пиксела) используется функция 0Ch прерывания 10h. Эта функция требует занесения в регистр AL кода цвета, в BH – номера видеостраницы, в CX – х-координаты выводимой точки в диапазоне 0...349, а в DX – у-координаты точки в диапазоне 0...639. Поскольку регистр CX используется, как счетчик шагов в цикле, для хранения х-координаты зарезервирован регистр SI.

Прямая горизонтальная линия в примере 53.1 рисуется путем вызова функции 0Ch в цикле, на каждом шаге которого значение у-координаты остается неизменным (175 в примере), а значение х-координаты увеличивается на единицу (предложение 13). После завершения цикла формирования изображения в программе предусмотрена остановка (предложения 16 и 17) для того, чтобы пользователь мог, оставаясь в графическом режиме, проанализировать результаты работы программы. Для остановки программы используется функция DOS 08h ввода одного символа с клавиатуры. Функция 08h, как уже отмечалось, не отображает введенный символ на экране и тем самым не искажает графическое изображение. Нажатие любой клавиши (кроме управляющих – Ctrl, Alt, Shift и др.) возобновляет выполнение программы.

В конце рассматриваемого фрагмента предусмотрено переключение видеоадаптера в стандартный текстовый режим с номером 03h (предложения 18...20). Если такое переключение не выполнить, видеоадаптер останется в графическом режиме, что может помешать правильному выполнению прикладных программ.

Рассмотрим кратко параметры вызова функции 0Ch прерывания 10h. В регистр BH заносится номер видеостраницы, на которую выводится данная точка. Графический адаптер EGA обеспечивает хранение и отображенис двух графических страниц. По умолчанию видимой (активной) является страница 0, однако рисовать изображение можно как на видимой, так и на невидимой страницс. Для переключения страниц предусмотрена функция 05h прерывания 10h.

В регистр AL заносится код цвета точки. В каждый момент времени изображение на экране может содержать только 16 цветов. Этот набор цветов, выводимых на экран (цветовая палитра), задается программно и может быть изменен. При загрузке компьютера устанавливается стандартная палитра, коды цветов которой были приведены в табл. 24.1.

Режим EGA, на примере которого мы рассмотрели принцип формирования графических изображений, можно использовать практически при любом видеоадаптере. Исключение составляют лишь монохромные адаптеры, имеющие специальное применение, а также устаревшие и почти не встречающиеся адаптеры CGA.

Практически все современные видеоадаптеры относятся к классу SVGA (Super Video Graphics Array). Хотя они и допускают использование режимов EGA и VGA, однако, как правило, применяются для вывода графических изображений со значительно лучшими характеристиками. Качество изображения, т. е. разрешение по вертикали и горизонтали, а также количество цветов определяется как характеристиками видеоадаптера, так и возможностями монитора. Все выпускаемые в настоящее время видеоадаптеры поддерживают режим TrueColor, при котором используется палитра из 16 млн цветов, а допустимое разрешение по вертикали и горизонтали зависит от объема памяти, установленной на видеоадаптере (видеопамяти). При этом следует не забывать, что монитор должен обеспечить выполнение режима, в котором заставляет его работать видеоадаптер. Вопросы, связанные с правильным выбором монитора, мы не обсуждаем, считая, что ваш монитор соответствует возможностям видеоадаптера.

Поскольку характеристики видеосистем, как и самих компьютеров, совершенствуются чрезвычайно стремительно, привести их все практически невозможно. Для того, чтобы проиллюстрировать возможности видеоадаптеров, в табл. 53.1 приведены режимы, реализуемые современными графическими платами серии ASUS AGP-V3800, в зависимости от установленной на них видеопамяти.

Таблица 53.1. Режимы, реализуемые графическими платами серии ASUS AGP-V3800

Разрешение по горизонтали, пиксел	*Разрешение по вертикали, пиксел*	*Объем видеопамяти, Мбайт*
480	640	8
600	800	8
768	1024	8
864	1152	8
1024	1280	8
1200	1600	8
1080	1920	16
1200	1920	16
1536	2048	32

К сожалению, для SVGA нет общепринятого стандарта, как для EGA или VGA. Хотя стандарт для SVGA предложен ассоциацией по стандартизации в видеоэлектронике (Video Electronics Standards Assotiation, VESA), в которую входят такие известные фирмы – разработчики аппаратного и программного обеспечения, как Intel, Microsoft, Philips Semiconductors, NVidia, Brook Tree, Cirrus Logic, Matrox Graphics, Phoenix Technologies, и ряд других, он поддерживается не всеми изготовителями видеоадаптеров.

Функции, определенные стандартом VESA, записываются непосредственно в ПЗУ адаптера. Они называются расширением прерывания BIOS 10h – VESA BIOS Extention или VBE. Для вызова функции VBE в регистр AH необходимо записать 4Fh, а в регистр AL -- номер функции. Однако функция может и не выполниться, если она отсутствует в вашей версии VBE. Может быть и так, что в VBE эта функция есть, но ее не

поддерживает аппаратура видеоадаптера. Если VBE поддерживает функцию, то в регистре AL возвращается значение 4Fh. Иначе возвращается другое число. В случае успешного выполнения функции в AL возвращается 0, а при ошибке – 1. Если в регистре AH возвращается 2h, это означает, что данную функцию не поддерживает аппаратура видеоадаптера.

Спектр характеристик видеоадаптеров SVGA очень широк, поэтому составление программы, формирующей графическое изображение и способной вывести его не с одним конкретным видеоадаптером, а с любым из заданной группы, особенно учитывая отсутствие стандарта, может представлять достаточно трудоемкую задачу. Функции VBE позволяют определить тип видеоадаптера, разрешенные графические режимы, установить требуемый графический режим и перевести монитор в энергосберегающий режим работы. Перечень графических режимов, поддерживаемых стандартом VBE 3.0, и номеров функций, обеспечивающих их установку, приведен в табл. 53.2. Полную и самую новую информацию о функциях VBE можно найти на сайте VESA www@vesa.org.

Рассмотрим пример работы с VBE. Для простоты предположим, что используемый нами режим поддерживается видеоадаптером, и посмотрим, как изменится приведенный выше фрагмент программы вывода на экран горизонтальной прямой. Выберем режим 103h (пример 53.2).

Таблица 53.2. Перечень графических режимов, поддерживаемых стандартом VBE 3.0

Номер функции	*Разрешение, пиксел*	*Количество цветов*
100h	640*400	256
101h	640*480	256
102h	800*600	16
103h	800*600	256
104h	1024*768	16
105h	1024*768	256
106h	1280*1024	16
107h	1280*1024	256
10Dh	320*200	32 K
10Eh	320*200	64 K
10Fh	320*200	16,8 M
110h	640*480	32 K
111h	640*480	64 K
112h	640*480	16,8 M
113h	800*600	32 K
114h	800*600	64 K
115h	800*600	16,8 M
116h	1024*768	32 K
117h	1024*768	64 K
118h	1024*768	16,8 M
119h	1280*1024	32 K
11Ah	1280*1024	64 K
11Bh	1280*1024	16,8 M
81FFh	Специальный режим, в котором обеспечивается доступ ко всей видеопамяти	

Сначала необходимо установить требуемый режим работы. Для этого воспользуемся функцией 02h. Загрузим в регистр AH номер функции, в регистр AL – номер подфункции, а в регистр BX – номер режима VESA и выполним прерывание BIOS 10h. Если графический режим установлен, то после выполнения прерывания в регистре AH возвращается 0. Поэтому в предложении 5 мы проверяем содержимое регистра AH и в случае неудачи выводим сообщение об ошибке – предложения 37... 39.

Перед выводом отрезка прямой нам требуется определить ее цвет. Поскольку мы установили режим 103h, в котором используется 256 цветов, то определим цвет с помощью функции 10h установки регистров палитры. Вызов этой функции выполняется в предложениях 13...19. В предложении 14 определяется номер подфункции, тоже 10h, которая позволяет непосредственно задать цвет для любого из 256 регистров таблицы цветов. Интенсивности красной, зеленой и синей составляющих задаются в регистрах DH, CH и CL. Поскольку мы решили вывести линию зеленого цвета с максимальной яркостью, то в регистр CH заносим максимальное значение $2^6-1=63$, а в остальные регистры – нули. Номер регистра таблицы цветов определяем в регистре BX.

Пример 53.2. Вывод на экран горизонтальной прямой в режиме 103h SVGA

```
;Установим графический режим 103h SVGA
        mov    AH,4Fh      ;(1)Функция вызова Video BIOS Extention
        mov    AL,02h      ;(2)Подфункция установки режима
        mov    BX,103h     ;(3)Графический режим SVGA 800x600x256
        int    10h         ;(4)Прерывание BIOS
        cmp    AH,0        ;(5)При ошибке
        jne    errmes1     ;(6)перейти на вывод сообщения
;Установим в регистре 150 таблицы цветов
;зеленый цвет максимальной яркости
        mov    AH,10h      ;(13)Функция управления регистрами палитры
        mov    AL,10h      ;(14)Подфункция установки регистра цветов
        mov    BX,150      ;(15)Номер регистра таблицы цветов(0-255)
        mov    DH,0        ;(16)Интенсивность красного цвета (6 бит)
        mov    CH,63       ;(17)Интенсивность зеленого цвета (6 бит)
        mov    CL,0        ;(18)Интенсивность синего цвета (6 бит)
        int    10h         ;(19)Прерывание BIOS
;Нарисуем прямую линию в цикле по координате x
        mov    SI,0        ;(20)Начальная x-координата
        mov    CX,800      ;(21)Число точек по горизонтали
line:   push   CX          ;(22)Сохраним его в стеке
        mov    AH,0Ch      ;(23)Функция вывода пиксела
        mov    AL,150      ;(24)Цвет зеленый
        mov    BH,0        ;(25)Видеостраница
        mov    CX,SI       ;(26)x-координата (переменная)
        mov    DX,300      ;(27)y-координата (константа)
        int    10h         ;(28)Вызов BIOS
        inc    SI          ;(29)Инкремент x-координаты
        pop    CX          ;(30)Восстановление счетчика шагов
        loop   line        ;(31)Цикл из CX шагов
;Остановим программу для наблюдения результатов ее работы
        mov    AH,08h      ;(32)Функция ввода с клавиатуры без эха
        int    21h         ;(33)Вызов DOS
;Переключим видеоадаптер назад в текстовый режим
        mov    AX,3        ;(34)Установка текстового режима
        int    10h         ;(35)Вызов BIOS
        jmp    output      ;(36)
errmes1:mov    AH,09h      ;(37)Функция вывода сообщения
        mov    DX,offset message1;(38)Смещение сообщения
        int    21h         ;(39)Вызов DOS
output:                    ;(40)Завершение программы
...
```

Остальные предложения практически идентичны соответствующим предложениям фрагмента, приведенного в примере 53.1, за исключением предложений 20, 21, 24 и 27. Поскольку линию будем рисовать с самого края и до конца экрана, в предложении 20 в регистр SI заносим 0, а в предложении 22 в регистр CX – 800. Чтобы расположить линию посередине экрана при выбранном разрешении в 600 точек по вертикали, в предложении 27 в регистр DX заносим 175. Для определения цвета в AL заносим номер регистра таблицы цветов, содержимое которого мы определили в предложениях 13...19, а именно 150.

Статья 54. **Динамическое управление памятью**

Как уже отмечалось ранее, при загрузке в память программы (как .COM, так и .EXE) DOS, независимо от фактического размера программы, выделяет ей по умолчанию всю наличную память. Такой алгоритм выделения памяти определяется (для программ типа .EXE) значением поля со смещением Ch в заголовке файла загрузочного модуля (см. табл. 32.2).

Корректная программа может освободить лишнюю для ее функционирования память и работать далее лишь в пределах закрепленного за ней адресного пространства. Это особенно важно для драйверов устройств и других программ, резидентных в памяти. Если драйвер после загрузки и инициации не освободит лишнюю память, система перестанет функционировать, так как некуда будет загружать пользовательские программы. С другой стороны, активной программе может на время потребоваться дополнительная память для размещения, например, данных, поступающих из измерительной аппаратуры. В этом случае программа может сделать запрос к DOS на выделение требуемого блока памяти, причем DOS в ответ сообщает об объеме свободной памяти, что позволяет программе приспособиться к конкретным условиям работы. Такой механизм важен для нормального функционирования программ, активно использующих оперативную память, так как в реальных условиях ввиду широкого использования резидентных программ (как системных, так и прикладных) объем свободной памяти может изменяться в широких пределах.

Программа, загруженная в память, включает три важных для программирования компонента: окружение, префикс программного сегмента и собственно программу, которая (в случае файла .EXE) может состоять из нескольких сегментов.

В процессе начальной загрузки DOS создает начальное окружение, в котором будут работать активизируемые программы, и прежде всего командный процессор COMMAND.COM. Окружение представляет собой область памяти, в которой в виде символьных строк записаны значения переменных, называемых переменными окружения. Формат окружения приведен на рис. 54.1.

```
'переменная_1 = значение_1', 0
'переменная_2 = значение_2', 0
'переменная_3 = значение_3', 0
...
'переменная_n = значение_n', 0, 0, 1, 0
'диск:\путь\имя_файла.расширение', 0
```

Рис. 54.1. Формат окружения (все числа занимают по 1 байту)

Имеется ряд переменных окружения, имена которых зарезервированы и известны системе, однако пользователь может включать в окружение и свои переменные для

использования их прикладными программами. Окружение служит для передачи программам (как системным, так и прикладным) требуемых параметров. Параметры (в виде значений определенных переменных окружения) заносятся в окружение с помощью системной команды SET. Системные и прикладные программы могут анализировать текущий состав окружения и извлекать из него относящиеся к ним параметры. Например, системная команда DIR ожидает найти в своем окружении (т. е. в окружении командного процессора COMMAND.COM) переменную DIRCMD с перечнем ключей и, если такая переменная имеется, команда DIR организует свой вывод в соответствии с указанными ключами.

Пусть в начале сеанса на машине мы выполнили команду

```
SET DIRCMD=/A:-D /O:-S
```

После этого команда DIR без указания ключей будет выводить на экран содержимое указанного каталога без подкаталогов (ключ /A:-D), упорядоченное по размеру файлов в порядке убывания размера (ключ /O:-S).

Механизм программного анализа окружения и извлечения параметров будет рассмотрен позже.

Размер окружения задается на этапе конфигурирования системы с помощью команды SET COMSPEC=, которая устанавливает местоположение и некоторые характеристики командного процессора. Например, команда

```
SET COMSPEC=C:\DOS_62\COMMAND.COM/P/E:400
```

говорит системе, что командный процессор находится в каталоге DOS_62, является резидентным (ключ /P) и работает в окружении размером 400 байт (ключ /E). Указанная команда позволяет увеличить размер начального окружения, что нужно в тех случаях, когда предполагается включать в окружение большое количество переменных.

Полная спецификация файла с программой, включаемая системой в окружение, позволяет, найдя в памяти окружение некоторой неизвестной программы, узнать имя этой программы (в самой загруженной в память программе, так же как и в файле загрузочного модуля, нет информации о ее имени).

Обычно DOS размещает окружение над программой вплотную к ней. Если, однако, при загрузке программы перед ней обнаруживается свободный участок (он мог возникнуть, если запущенная перед этим резидентная программа после загрузки освободила свое окружение), окружение данной программы размещается в этой "дырке" в памяти. В любом случае сегментный адрес окружения помещается системой в PSP программы. Поскольку окружение всегда начинается на границе параграфа, его местоположение однозначно описывается сегментным адресом, который можно найти в PSP в слове со смещением 2Ch (см. табл. 32.1).

DOS выделяет оперативную память участками произвольной длины, которые обычно называют блоками. Размер блока задается в параграфах и в принципе может иметь значение от 0 до FFFFh, т. е. все адресное пространство может быть отдано одному блоку. DOS ведет учет занятой и свободной памяти с помощью 16-байтовых структур – блоков управления памятью (Memory Control Block, MCB). Каждый MCB располагается в памяти непосредственно перед тем блоком, к которому он относится. Формат MCB приведен на рис. 54.2.

Байты	0	1...2	3...4	5...7	8...15
	Тип	Адрес владельца	Размер блока	Резерв	Имя программы, если блок – программа

Значения типа: 'M' = 4Dh – промежуточный блок
'Z' = 5Ah – последний блок

Рис. 54.2. Формат блока управления памятью MCB

Для динамического управления памятью используются три функции DOS (табл. 54.1).

Таблица 54.1. Функции управления памятью

Номер функции	*Назначение*
48h	Выделить блок памяти
49h	Освободить блок памяти
4Ah	Изменить размер выделенного блока памяти

С помощью функции 48h программа может затребовать у DOS до 1 Мбайт памяти, хотя практически разумно получать память сегментами по 64 Кбайт. Размер требуемого блока (в параграфах) указывается в регистре BX. В случае успешного завершения функции сегментный адрес выделенного блока памяти возвращается в AX; программа, переслав этот адрес в сегментный регистр данных (обычно ES), может работать с выделенной памятью, которая с точки зрения структуры программы представляет собой дополнительный сегмент данных. Если DOS не смогла выделить память (о чем говорит установленный флаг CF), в регистре BX возвращается число свободных параграфов и программа может проанализировать это значение с целью определения дальнейшей стратегии.

При каждом выделении блока памяти DOS создает блок управления памятью, размещаемый непосредственно перед выделяемым блоком. Следует заметить, что работа с блоками управления памятью является исключительной прерогативой DOS; случайное разрушение блока приводит к выдаче сообщения "Ошибка распределения памяти" и останову системы.

Для освобождения блока памяти, выделенного программе с помощью функции 48h, используется функция 49h. Нельзя освободить память, которой программа не обладает. Нельзя также освободить только часть выделенной памяти (для этого используется функция изменения размера блока 4Ah).

Для изменения размера блока памяти, ранее выделенного программе функцией 48h, а также для изменения собственного размера (как увеличения, так и уменьшения) предусмотрена функция 4Ah. Новый размер (в параграфах) изменяемого блока памяти (с точки зрения структуры программы – сегмента или группы сегментов) передается в регистре BX, а сегментный адрес этого блока – в регистре ES. Функция 4Ah обычно используется для сокращения размера программы до реально необходимого (вспомним, что при загрузке DOS выделяет программе всю наличную память). В этом случае программе требуется определить свой реальный размер, а также свой базовый сегментный адрес. Любая программа (и .EXE, и .COM) начинается с префикса программного сегмента, причем при загрузке программы базовый адрес PSP находится в регистрах ES и DS (а для программ .COM также и в регистрах CS и SS). Если программа явным образом не модифицирует содержимого ES, то этот регистр уже оказывается настроен должным образом для вызова функции 4Ah.

Методика определения размера программы зависит от типа программы (.EXE или .COM), а также от того, надо ли сохранить в памяти всю программу или только ее часть. Если в программе .EXE, состоящей из трех сегментов – программного, данных и стека, предусматривается освобождение лишней памяти, то для определения размера программы можно включить в программу фиктивный пустой сегмент, расположив его в самом конце программы, после всех остальных сегментов. Учитывая, что при загрузке программы ее начальный адрес (т. е. адрес PSP) заносится в регистр ES, для определения размера программы потребуются следующие строки:

```
text    segment
        mov   AX,fict       ;Сегментный адрес конца программы
        mov   BX,ES         ;Сегментный адрес начала программы
        sub   AX,BX         ;AX=размер программы в параграфах
        ...
text    ends
data    segment
        ...
data    ends
stack   segment stack
        ...
stack   ends
fict    segment             ;Фиктивный сегмент
fict    ends                ;нулевой длины
```

Размер односегментой программы .COM определяется иначе. Он равен разности значений счетчика текущего адреса в конце и начале программы плюс размеру PSP. Однако при использовании функции 4Ah в программе .COM необходимо иметь в виду, что при загрузке программы весь остаток сегмента размером 64 Кбайт, не занятый собственно программой и ее данными, отдается стеку, а указатель стека инициализируется числом FFFEh и указывает, таким образом, на последнее слово сегмента. Для того чтобы сократить размер стека до разумной величины, перед освобождением лишней памяти указатель стека следует переместить в конец специально выделенной области стека:

```
text    segment
        org   100h
main    proc
        ...
        mov   SP, offset newstk
main    endp
        ...   ;Данные
        dw 64 dup (?)       ;Область стека
newstk=$
text    ends
```

Размер программы в параграфах будет равен (newstk-main+100h+0Fh)/16. Здесь к вычисленному размеру программы добавлен размер PSP (100h байт) и число Fh для округления до следующего параграфа. Деление всей суммы на 16 дает число параграфов.

Рассмотрим пример программы, которая, предварительно освободив всю наличную память, выделяет себе блок памяти размером 64 Кбайт и заполняет его некоторым содержимым (пример 54.1). Для контроля правильности работы программы выведем содержимое заполненного блока в файл на диске.

Пример 54.1. Динамическое выделение памяти

```
text    segment
        assume CS:text,DS:data
```

```
myproc  proc
        mov     AX,data
        mov     DS,AX
;Программа .EXE занимает всю память. Освободим ее
        mov     BX,abcd      ;Сегментный адрес конца программы
        mov     AX,ES        ;Сегментный адрес начала программы
        sub     BX,AX        ;Размер программы в параграфах
        mov     AH,4Ah       ;Функция изменения размера блока
        int     21h          ;Освободили лишнюю память
;Теперь выделим 64 Кбайт
        mov     AH,48h       ;Функция выделения памяти
        mov     BX,1000h     ;1000h параграфов = 64 Кбайт
        int     21h
        mov     allocseg,AX ;Адрес выделенного блока
;Заполним выделенный блок памяти каким-либо символом
        mov     ES,AX        ;Настроим на него ES
        xor     DI,DI        ;ES:DI → начало выделенного блока
        mov     CX,0FFFFh    ;65535 - счетчик байтов
        mov     AL,'@'       ;Заполняющий символ
        cld                  ;Заполнять вперед
rep     stosb                ;Заполнение
        mov     byte ptr ES:[DI],'#';Заполним последний байт
                             ;массива для контроля
;Создадим файл и сохраним возвращенный дескриптор в ячейке handle
        mov     AH,3Ch
        mov     CX,0
        mov     DX,offset fname
        int     21h
        mov     handle,AX
;Запишем заполненный блок в файл. Поскольку за один раз можно
;вывести в файл не более 64 К-1 байт, запишем два раза по 32 К
;Сначала первую половину массива
        mov     AH,40H       ;Функция записи
        mov     BX,handle    ;Дескриптор открытого файла
        mov     CX,32768     ;Выведем половину массива
        mov     DS,allocseg ;Настроим DS на выделенный блок
        xor     DX,DX        ;DS:DX → на выделенный блок
        int     21h
;Затем вторую половину массива. Почти все регистры уже настроены
        mov     AH,40h
        mov     DX,32768     ;Продолжим запись с этого байта
        int     21h
;Освободим выделенный блок памяти. ES уже указывает на него
        mov     AH,49h       ;Функция освобождения памяти
        int     21h
;Завершим программу
        mov     AX,4C00h
        int     21h
myproc  endp
text    ends
data    segment
handle  dw      0            ;Дескриптор
allocseg dw     0            ;Сегментный адрес выделенного блока
fname   db      'BIGFILE.DAT',0 ;Имя файла для записи массива
data    ends
stk     segment stack
        dw      128 dup (?)
stk     ends
abcd    segment              ;Фиктивный сегмент для определения
abcd    ends                 ;конца программы
        end     myproc
```

В результате выполнения приведенной выше программы в текущем каталоге текущего диска будет создан файл с именем BIGFILE.DAT длиной точно 64 Кбайт, заполненный кодами символа @. Самый последний байт этого файла будет содержать символ #.

В заключение этой статьи рассмотрим трюк, который вряд ли имеет практический смысл, но полезен как средство усвоения пройденного материала. Как известно, в ячейке PSP со смещением 0Ah система хранит адрес возврата в командный процессор COMMAND.COM (см. табл. 32.1). Дальний косвенный переход через эту ячейку приведет к передаче управления в командный процессор без каких-либо завершающих действий, в частности без освобождения памяти, занимаемой программой. Правда, в DOS встроены средства исправления этой ошибки и восстановления системы после такого "завершения", однако лучше все-таки освободить память явным образом. В примере 54.2 вызовом функции 49h сначала освобождается блок памяти с самой программой, а затем после переноса в ES из PSP сегментного адреса окружения – блок памяти окружения. Наконец, командой дальнего косвенного перехода управление передается командному процессору.

Пример 54.2. Завершение программы "вручную"

```
;При запуске программы DS и ES указывают на PSP
        mov    AH,49h        ;(1)Функция освобождения памяти
        int    21h           ;(2)Освободим всю программу
        mov    AH,49h        ;(3)Функция освобождения памяти
        mov    ES,ES:2Ch     ;(4)ES → окружение
        int    21h           ;(5)Освободим окружение программы
        jmp    dword ptr DS:0Ah;(6)Передадим управление в COMMAND.COM
```

В приведенном примере есть тонкость, на которую стоит обратить внимание. Результатом выполнения предложения 2 является освобождение памяти, занимаемой программой. Однако мы как ни в чем не бывало продолжаем выполнять эту программу (предложения 3...6). Это возможно потому, что система не затирает освобождаемую память. Освобождение блока памяти приводит к перестройке блоков MCB, однако коды, составляющие программу, остаются в памяти и их можно продолжать выполнять. Затерты эти коды будут лишь после того, как на то же место памяти будет загружена очередная программа.

Статья 55. **Динамическое управление процессами**

Программы загружаются в память для выполнения с помощью функции DOS Exec (Int 21h, функция 4Bh), которая играет роль системного загрузчика. Если пользователь запускает программу, вводя командную строку с клавиатуры, то функцию Exec вызывает командный процессор COMMAND. В других случаях функция Exec может быть вызвана загруженной и выполняемой программой, в том числе пользовательской. Такая динамическая загрузка и запуск дочерних программ позволяют организовать иерархические программные комплексы, в которых родительский процесс, в зависимости от конкретных условий, инициирует те или иные дочерние процессы, а те, в свою очередь, могут вызывать к жизни процессы следующего уровня и т. д. При этом надо иметь в виду, что система MS-DOS не является системой реального времени и не поддерживает параллельных процессов. В иерархическом программном комплексе все его составляющие выполняются только поочередно, друг за другом.

Как уже отмечалось, программа, загруженная в память, занимает, как правило, 4 блока памяти: блок окружения программы с принадлежащим ему блоком управления памятью MCB и блок собственно программы с ее MCB; самостоятельным элементом программы является ее PSP (рис. 55.1).

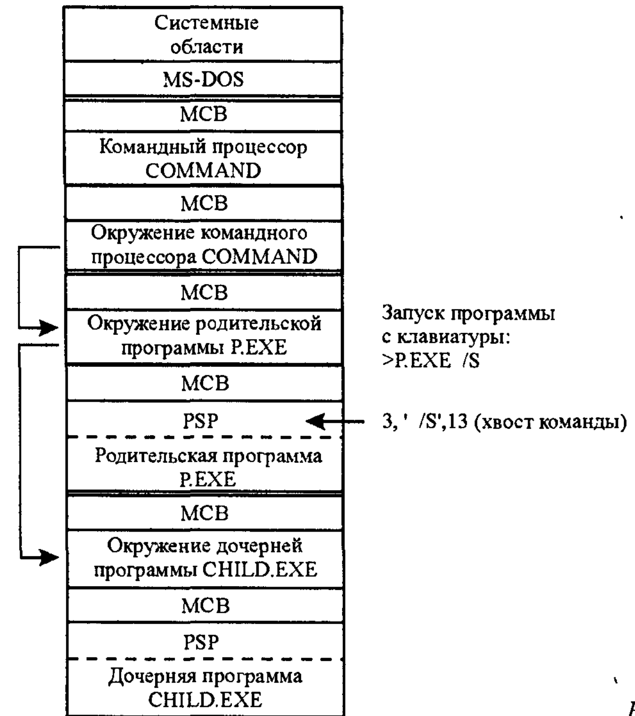


Рис. 55.1. Запуск дочернего процесса

Окружение для командного процессора, создаваемое в процессе начальной загрузки, в простейшем случае содержит переменные COMSPEC, PROMPT и PATH, которые заносятся в окружение из файла AUTOEXEC.BAT, и может иметь, например, следующий вид:

```
COMSPEC=C:\DOS\COMMAND.COM
PROMPT=$P$G
PATH=C:\; C:\DOS; C:\TOOLS
```

Первая строка описывает путь к файлу командного процессора COMMAND.COM. Системные и прикладные программы, загружаясь в память, затирают транзитную часть командного процессора, и после завершения такой программы командный процессор надо снова загрузить с диска. Система (конкретно – резидентная часть COMMAND.COM) берет имя и местонахождение файла с командным процессором из значения переменной окружения COMSPEC.

Строка с переменной PROMPT задает вид системного запроса, который в приведенном примере состоит из полного пути к текущему каталогу и изображения стрелки.

Строка с переменной PATH определяет пути, которые автоматически просматриваются DOS при поиске на диске запущенной пользователем программы.

Пользователь может включить в окружение строки определения дополнительных переменных с помощью команды SET. При вводе команды с клавиатуры значения этих переменных будут действовать только в данном сеансе работы на компьютере до его перезагрузки. Если же включить команду SET в файл AUTOEXEC.BAT, она будет автоматически выполняться каждый раз при загрузке системы и значения дополнительных переменных будут определены всегда.

После того как выполнена начальная загрузка DOS, активной программой (текущим процессом) становится командный процессор COMMAND.COM, который фактически ничего не делает, ожидая ввода команды с клавиатуры. Получив команду оператора на запуск программы (например, программы P.EXE на рис. 55.1), COMMAND.COM, активизируя эту программу (с помощью функции DOS Exec), передает ей свое окружение. Практически это осуществляется путем копирования строк окружения в новую область памяти, которая обычно располагается перед загружаемой программой. Если загруженная программа по ходу своего выполнения в свою очередь активизирует дочерний процесс следующего уровня (с помощью той же функции Exec), она еще раз копирует свое окруженис, передавая его запускаемой ею программе. При этом родительский процесс может передать дочернему свое окружение в неизменном виде, а может включить в него новую информацию. Таким образом, каждая программа, загруженная в память, имеет собственную копию окружения, причем эти копии могут совпадать, а могут и различаться. После завершения программы и возврата управления родительскому процессу окружение дочерней программы уничтожается.

Многие программы не используют переменные окружения и передача им окружения носит формальный характер. С другой стороны, с помощью этого механизма родительские процессы могут передавать дочерним значительные объемы информации (максимальный размер блока окружения составляет 32 Кбайт).

Второй структурой данных, формируемой функцией Exec при активизации вызываемой программы, является ее префикс PSP, всегда располагаемый в самом начале загружаемой программы. В слово PSP со смещением 2Ch DOS помещает сегментный адрес окружения программы, а в область, начинающуюся с адреса 80h, – параметры командной строки, или "хвост команды" (/S на рис. 55.1). В байте по адресу 80h находится длина параметров команды без учета завершающего команду кода 13, а затем располагаются сами параметры в виде символьной строки. С помощью хвоста командной строки запускаемой программе часто передаются ключи, управляющие ее работой, а также имена рабочих файлов. Так, программа P.EXE могла быть запущена командой

```
>P.EXE A:\WORKFILE.001/S
```

в которой программе, помимо ключа /S, определяющего режим ее работы, передается еще и спецификация рабочего файла WORKFILE.001. Если пользовательскую программу предполагается запускать с передачей ей каких-то параметров, то в программе должны быть строки извлечения хвоста команды из PSP и его анализа.

Для загрузки и выполнения дочерней программы родительская программа должна вызвать прерывание Int 21h с функцией 4Bh при следующем состоянии регистров:

```
AH=4Bh;
AL=00h (подфункция загрузки дочерней программы);
DS:DX=адрес строки со спецификацией файла с дочерней программой;
ES:BX=адрес блока параметров.
```

Блок параметров служит для передачи функции Exec необходимой информации и включает в себя 4 адреса: окружения, параметров командой строки и двух блоков управления файлами FCB. Блоки управления файлами в настоящее время не используются, и их адреса следует заменить нулями.

Блок параметров и связанные с ним структуры данных могут выглядеть следующим образом:

```
parmblk dw     envirseg     ;Сегмент окружения
        dd     cmdtail      ;Адрес хвоста команды
        dd     0,0          ;Адреса FCB
chname  db 'CHILD.EXE',0    ;Имя дочернего процесса
cmdtail db 9,' FILE.TXT',13;Хвост команды
envirseg segment 'ENVIR'    ;Сегмент окружения
        db     'WORKFILE=A:\FILES',0
envirseg ends
```

Адрес окружения составляет одно слово и содержит только сегментный адрес, поскольку блок окружения всегда начинается на границе сегмента. Если дочерний процесс должен обладать специфическим окружением, в родительской программе предусматривается и заполняется необходимой информацией соответствующий сегмент. Если же дочерний процесс должен наследовать окружение родительского, что обычно и имеет место, то специальный сегмент не создается, а в качестве адреса окружения в блоке параметров указывается 0.

В приведенном выше примере сегмент окружения заполнен в родительской программе явным образом и содержит индивидуальную переменную дочернего процесса WORKFILE, служащую, например, для передачи дочерней программе CHILD.EXE имени каталога со вспомогательными файлами. В этом случае окружение системного процесса будет очень коротким и включать лишь эту индивидуальную переменную (плюс имя дочерней программы). Для использования информации, передаваемой через окружение, дочерняя программа извлекает из слова 2Ch в PSP физический адрес окружения и ищет затем в блоке окружения интересующие ее переменные.

Хвост команды используется для передачи дочерней программе параметров вызова, чаще всего имен рабочих файлов, а также ключей, определяющих режим работы программы. В процессе загрузки дочерней программы DOS перешлет хвост команды из блока параметров функции Exec на его "законное" место – в префикс дочерней программы, где он будет располагаться начиная с адреса 80h. Для извлечения его из PSP и анализа в дочерней программе должны быть предусмотрены соответствующие программные строки.

В случае запуска программы с клавиатуры параметры командой строки вводятся вслед за именем запускаемой программы вручную; при программной (динамической) активизации дочернего процесса эти параметры определяются в родительской программе, как это показано в приведенном выше программном фрагменте.

В блоке параметров хвост команды должен иметь тот же формат, что и в PSP, т. е. начинаться с байта – счетчика, за которым следует символьная строка параметров. Завершается строка символом возврата каретки (13), который не входит в счет байтов. Если в конкретном вызове дочерней программы параметры ей не передаются, в блоке параметров следует вместо адреса хвоста команды указать 0.

В простейшем варианте (наследуется окружение родителя, а хвост команды и FCB отсутствуют) структуры данных для вызова функции Exec приобретают следующий вид:

```
parmblk dw 7 dup (0)
chname  db 'CHILD.EXE',0
```

Перед активизацией дочернего процесса родительская программа должна освободить достаточный для загрузки дочерней программы объем памяти. При нехватке памяти или при отсутствии запрашиваемого файла функция Exec возвращает установленный бит CF.

Все файлы и устройства, открытые родительской программой с помощью выделения дескрипторов, дублируются в дочерней программе, т. е. дочерняя программа наследует все открытые дескрипторы родительской. Таким образом, обе программы могут совместно использовать одни и те же файлы, при этом операции ввода-вывода, выполняемые в дочерней программе, отражаются на состоянии дескрипторов родительской.

Активизировав дочерний процесс с помощью функции Exec, DOS приостанавливает выполнение родительской программы до завершения дочерней. Дочерняя программа может, в свою очередь, загружать другие программы с помощью той же самой процедуры, передавая управление через несколько уровней, пока в системе не исчерпается память.

DOS позволяет дочернему процессу передать в вызвавший его родительский процесс код возврата (завершения). Этот код может принимать значение от 0 до 255. Обычно 0 считается кодом успешного завершения, а все остальные значения говорят об ошибках. Значение кода возврата формируется в дочерней программе по мере ее выполнения и передается DOS с помощью функции 4Ch, которой обычно завершается любая, в том числе и дочерняя программа, чтобы передать управление родительскому процессу:

```
mov   AH,4Ch      ;Функция завершения процесса
mov   AL,errcode  ;Код завершения
int   21h         ;Возврат в родительский процесс
```

DOS сохраняет полученный ею код возврата завершившегося процесса в области текущих данных (см. табл. 44.1, смещение 14h) до тех пор, пока его не заменит код завершения очередного процесса.

Если родительской программе требуется получить код завершения дочерней, то она сразу же после возврата в нее управления из дочерней должна выполнить функцию DOS 4Dh, извлекающую код завершения из соответствующей системной ячейки и передающей его вызывающей программе.

Рассмотрим пример иерархического программного комплекса, в котором родительская программа с помощью функции Exec выполняет динамический запуск дочерней (как говорят, "порождает дочерний процесс") (пример 55.1). При подготовке этого примера следует иметь в виду, что выполнимый файл дочерней программы должен иметь именно то имя, на которое сделана ссылка в блоке параметров функции Exec родительской программы. В нашем примере для дочерней программы принято имя CHILD.EXE. Имя родительской программы никакого значения не имеет. Для того чтобы проиллюстрировать возврат в родительскую программу кода завершения дочерней, в последней предусмотрена операция открытия файла с (произвольным) именем FILE.TXT. Если файл открывается успешно, дочерняя программа при своем завершении возвращает код 0; если указанный файл на диске отсутствует, возвращается код 1. Родительская программа анализирует полученный ею код возврата и выводит соответствующие сообщения.

Пример 55.1. Иерархический программный комплекс. Родительская программа

```
;В сегменте команд
myproc  proc
        mov   AX,data
        mov   DS,AX
;Выведем сообщение msg1 о запуске родительского процесса
        mov   AH,09h
        mov   DX,offset msg1
        int   21h
;Программа .EXE занимает всю память. Освободим ее
        mov   BX,abcd      ;Сегментный адрес конца программы
        mov   AX,ES        ;Сегментный адрес начала программы
        sub   BX,AX        ;Размер программы в параграфах
        mov   AH,4Ah       ;Функция изменения размера блока
        int   21h
;Запустим дочерний процесс
        push  DS           ;Настроим ES на
        pop   ES           ;сегмент данных
        mov   AH,4Bh       ;Функция Exec
        mov   AL,0         ;Подфункция запуска программы
        mov   BX,offset parmblk;Адрес блока параметров
        mov   DX,offset chname;Адрес имени дочерней программы
        int   21h
        jc    errexec      ;Ошибка запуска
;Проанализируем код возврата из дочернего процесса
        mov   AH,4Dh       ;Функция получения кода возврата
        int   21h          ;из дочерней программы
        cmp   AL,1         ;Наш код ошибки?
        je    errchild     ;Да
;Выведем сообщение msg2 об успешном завершении дочернего процесса
        mov   AH,09h
        mov   DX,offset msg2
        int   21h
;Завершение программы
outprog:mov   AX,4C00h     ;Функция завершения, код
        int   21h          ;завершения = 0
;Выведем сообщение msg3 об ошибке при выполнении дочернего процесса
errchild:mov  AH,09h
        mov   DX,offset msg3
        int   21h
        jmp   outprog
;Выведем сообщение msg4 об ошибке при запуске дочернего процесса
errexec:mov   AH,09h
        mov   DX,offset msg4
        int   21h
        jmp   outprog
myproc  endp
;В сегменте данных
chname  db 'CHILD.EXE',0;Имя дочерней программы
parmblk dw    7 dup (0)
msg1    db '*P* Родительский процесс запущен',10,13,'$'
msg2    db '*P* Дочерний процесс отработал нормально',10,13,'$'
msg3    db '*P* Дочерний процесс завершился с ошибкой',10,13,'$'
msg4    db '*P* Дочерний процесс не активизирован',10,13,'$'
;Последний фиктивный сегмент программы для определения ее размера
abcd    segment
abcd    ends
```

Пример 55.1 (продолжение). Иерархический программный комплекс. Дочерняя программа.

```
;В сегменте команд
;Выведем сообщение msg1 о запуске дочернего процесса
        mov     AH,09h
        mov     DX,offset msg1
        int     21h
;Сделаем попытку открыть файл
        mov     AH,3Dh          ;Функция открытия файла
        mov     AL,0            ;Доступ для чтения
        mov     DX,offset fname         ;Адрес имени файла
        int     21h
        jnc     ok              ;Переход, если CF=0
;Обработаем ошибку открытия файла
        mov     errcode,1       ;Код родительского процесса
;Выведем диагностическое сообщение msg2
        mov     AH,09h
        mov     DX,offset msg2
        int     21h
        jmp     finend
;Завершим дочерний процесс
;Сначала выведем сообщение msg3 о нормальной работе
ok:     mov     AH,09h
        mov     DX,offset msg3
        int     21h
;И завершим программу с передачей родителю кода завершения
finend: mov     AH,4Ch          ;Функция завершения
        mov     AL,errcode      ;Код возврата
        int     21h
;В сегменте данных
msg1    db '*Д* Дочерний процесс запущен',10,13,'$'
msg2    db '*Д* Файл не открылся!',10,13,'$'
msg3    db '*Д* Файл открылся. Дочерний процесс завершается',10,13,'$'
fname   db 'FILE.TXT',0
errcode db 0
```

В приведенном примере предусмотрен анализ выполнения каждого этапа с выводом на экран соответствующего диагностического сообщения. Это дает возможность, запуская родительскую программу в различных вариантах, наглядно судить о ходе работы всего комплекса. Рассмотрим несколько вариантов операционной среды, в которой выполняется наш программный комплекс, с указанием последовательности выводимых диагностических сообщений.

1. В текущем каталоге диска имеются все составляющие программного комплекса, а именно родительская программа 55-01.EXE, дочерняя программа, загрузочный файл которой получил имя CHILD.EXE, а также рабочий файл FILE.TXT (с произвольным содержимым или даже без такового). В этом случае на экран будет выведена такая последовательность сообщений:

```
*Р* Родительский процесс запущен',10,13,'$'
*Д* Дочерний процесс запущен',10,13,'$'
*Д* Файл открылся. Дочерний процесс завершается',10,13,'$'
*Р* Дочерний процесс отработал нормально',10,13,'$'
```

2. В текущем каталоге отсутствует дочерняя программа CHILD.EXE. Последовательность сообщений этого варианта:

```
*Р* Родительский процесс запущен',10,13,'$'
*Р* Дочерний процесс не активизирован',10,13,'$'
```

3. В текущем каталоге имеются обе программы, но отсутствует рабочий файл FILE.TXT, что приводит к ошибке при выполнении дочерней программы. Последовательность сообщений:

```
*Р* Родительский процесс запущен',10,13,'$'
*Д* Дочерний процесс запущен',10,13,'$'
*Д* Файл не открылся!',10,13,'$'
*Р* Дочерний процесс завершился с ошибкой',10,13,'$'
```

Частным, но практически важным случаем активизации дочернего процесса является вызов из родительской программы второй копии командного процессора COMMAND.COM. Это дает возможность выполнять, не прерывая текущей программы, любые команды DOS, например копирования файлов, проверки качества или форматирования дискеты и т. д. Такая возможность предусматривается сегодня практически во всех коммерческих прикладных программах.

Для вызова с помощью функции Exec командного процессора следует указать полную спецификацию его файла, которая обычно извлекается из окружения текущего процесса, хотя может быть задана и непосредственно в полях данных программы.

При необходимости можно вызвать командный процессор с именем некоторой программы или команды DOS. Это имя следует передать дочернему процессу (программе COMMAND.COM) в хвосте команды, начав его с ключа /C (как это делается при вызове команды DOS COMMAND, служащей для той же цели). Хвост команды может в этом случае иметь следующий вид:

```
cmdtail db 13,' /C FORMAT A:',13
```

Здесь после загрузки второй копии командного процессора автоматически выполняется команда DOS FORMAT A:. Таким же образом можно выполнить и любую другую команду DOS или прикладную программу. Чаще, однако, командный процессор вызывается без ключа C и без имени конкретной команды. В этом случае командный процессор, получив управление, ожидает ввода команд DOS с клавиатуры. Пользователь может неограниченное время работать с DOS, вызывая любые команды DOS или прикладные программы. Для возврата в родительский процесс следует ввести команду EXIT.

Раздел пятый

АРИФМЕТИЧЕСКИЙ СОПРОЦЕССОР

Статья 56. **Основы работы с арифметическим сопроцессором**

Как, возможно, вы успели уже заметить, все рассмотренные программы предназначались для обработки символьной информации или выполнения операций над множеством целых чисел. Для того чтобы выполнить операцию над действительным числом, имеющим целую и дробную части, необходимо использовать арифметический процессор. Арифметические процессоры, которые обычно называются сопроцессорами, обрабатывают информацию, представленную в формате действительных чисел. Наличие сопроцессора позволяет значительно ускорить работу программ, выполняющих расчеты с высокой точностью, тригонометрические вычисления и обработку информации, которая должна быть представлена в виде действительных чисел.

Начиная с модели i486DX арифметический процессор располагается на том же кристалле, что и основной процессор. Тем не менее в дальнейшем, говоря о командах или особенностях арифметического процессора, мы будем называть его сопроцессором вне зависимости от того, каким образом он реализован.

Прежде чем перейти к составлению программ, остановимся на программной модели сопроцессора, которая показана на рис. 56.1.

ST	Верхнее слово стека
ST(1)	
ST(2)	
ST(3)	
ST(4)	
ST(5)	
ST(6)	
ST(7)	

Рис. 56.1. Программная модель сопроцессора

Программисту доступны 8 регистров общего назначения, обозначаемых ST(0)... ...ST(7), и 5 нечисловых регистров, которые будут рассмотрены позже. Десятибайтовые регистры ST(0)...ST(7) используются как стек, что подчеркивается их обозначением ST (STack). Они предназначены для хранения операндов и результатов арифметических операций. Первый из этих регистров, наиболее часто используемый, обычно обозначается просто ST.

Вне зависимости от формата данные, передаваемые на обработку в сопроцессор, преобразуются во внутреннее представление, занимающее 80 бит, и записываются в один из регистров общего назначения. После завершения обработки результат может

быть переслан в память, а перед этим преобразован в необходимый формат. Нечисловые регистры используются преимущественно для управления работой сопроцессора.

Для использования сопроцессора надо лишь включить в программу специальные команды – команды сопроцессора. Проиллюстрируем вышесказанное на примере сложения двух целых чисел, занимающих одно слово (пример 56.1). Заметим, что данные имеют такие же форматы, как и при работе с основным процессором.

Пример 56.1. Сложение на сопроцессоре целых чисел

```
text    segment
assume  cs:text,ds:data
myproc  proc
        mov    AX,data      ;Инициализируем
        mov    DS,AX        ;регистр DS
        fild   x            ;Загрузим 1-й операнд из ячейки x в ST
        fild   y            ;2-й операнд из y в ST,1-й операнд в ST(1)
        fadd                ;Сложим: сумма в ST, ST(1) освобождается
        fistp  z            ;Преобразуем ST в целый формат и в память
        mov    ax,4C00h
        int    21h
myproc  endp
text           ends
data segment
x       dw     1            ;Первый операнд
y       dw     2            ;Второй операнд
z       dw     ?            ;Результат
data    ends
end     myproc
```

В результате выполнения данной программы в поле z будет записано число 3. Убедиться в этом можно, используя отладчик в режиме пошагового выполнения (работу с отладчиком мы рассмотрим в одной из следующих статей) или дополнив программу фрагментом, обеспечивающим вывод содержимого поля z.

Следует иметь в виду, что использование сопроцессора исключительно для операций над целыми числами нецелесообразно, так как эти действия он выполняет медленнее, чем центральный процессор. Это происходит из-за того, что перед выполнением операции в сопроцессоре числа всегда преобразуются во внутреннюю, действительную форму представления и реально операции выполняются над действительными числами. Так, например, команда сложения содержимого регистра AX с операндом, находящимся в памяти

```
        add    AX,mem
```

выполняется за 24+n машинных тактов, а схожая команда сопроцессора

```
        fiadd  mem
```

за 120+n тактов, где n – количество тактов, необходимое для вычисления адреса операнда mem. При записи в память производится обратное преобразование с округлением, способ которого можно выбрать.

В приведенной выше программе использованы три команды сопроцессора: fild (float integer load, загрузка целого числа в формате плавающей точки), fadd (float add, сложение целых чисел в формате плавающей точки) и fistp (float integer store and pop, запись числа с плавающей точкой в формате целого и выталкивание из стека). Рассмотрим, как будет изменяться содержимое стека сопроцессора и полей данных программы в процессе выполнения этих команд (рис. 56.2).

Команда	Содержимое стека после выполнения этой команды	Содержимое полей данных
mov DS,AX	ST не определено ST(1) не определено ST(2) не определено	x 1 y 2 z ?
fild x	ST 1 ST(1) не определено ST(2) не определено	x 1 y 2 z ?
fild y	ST 2 ST(1) 1 ST(2) не определено	x 1 y 2 z ?
fadd	ST 3 ST(1) не определено ST(2) не определено	x 1 y 2 z ?
fistp x	ST не определено ST(1) не определено ST(2) не определено	x 1 y 2 z 3

Рис. 56.2. Содержимое полей данных и стека сопроцессора при выполнении примера 56.1

Первая команда fild x пересылает значение x из памяти в сопроцессор. После ее выполнения в регистре ST оказывается число из поля x. Выполнение следующей команды fild y приводит к тому, что содержимое поля y загружается в вершину стека ST. Загруженная ранее единица будет протолкнута в ST(1).

Команда fadd, в которой отсутствует явное определение данных, выполняет сложение чисел, расположенных в двух верхних регистрах стека ST и ST(1), и записывает результат в ST. Содержимое ST(1) после выполнения этой команды становится неопределенным, и в него теперь можно загружать новые числа. Учтите, что загрузка новых чисел в уже занятый регистр сопроцессора не допускается. Если вы попробуете это сделать, возникнет исключение.

Команда fistp выполняет запись результата в память и выгрузку регистра ST. Она нужна для преобразования полученного результата из внутреннего представления в целый формат и пересылки его в память. Как видно из рис. 56.2, в процессе выполнения этой команды содержимое ST выталкивается из стека. Если по ходу программы требуется переписать содержимое вершины стека ST в память и оставить стек без изменений, то следует использовать команду fist (float integer store, запись числа в формате плавающей точки в память в формате целого).

Аналогично производится сложение и вычитание целых чисел, для представления которых отводится 4 или 8 байт, и вещественных чисел, занимающих в памяти, в зависимости от требуемой точности, 4, 8 или 10 байт. После выполнения операций над

числами их можно записать в память в любом из определенных для сопроцессора форматов, используя соответствующие команды (список команд арифметического сопроцессора приведен в Приложении 3).

Статья 57. **Работа с действительными числами**

Начнем с фрагмента программы, иллюстрирующего сложение действительных чисел (пример 57.1). В нем вместо команд fild и fistp, которые мы применяли для сложения целочисленных данных, используются команды fld и fstp (вместе с командой fadd), а для хранения действительных чисел зарезервировано по 4 байта.

Пример 57.1. Сложение действительных чисел

```
        fld   x         ;Загрузим 1-й операнд в стек сопроцессора
        fld   y         ;Загрузим 2-й операнд в стек сопроцессора
        fadd            ;Сложим их
        fstp  z         ;Перешлем результат в память
;Поля данных
x       dd    1.0       ;Действительные константы должны быть
y       dd    2.5       ;описаны в формате с десятичной точкой
z       dd    (?)       ;например, 1.0. Описание 1 недопустимо
```

Напомним, что сопроцессоры специально предназначаются для обработки действительных чисел. Поэтому в стеке сопроцессора все числа представляются в действительном формате. Обычный, используемый по умолчанию формат представления чисел в сопроцессоре показан на рис. 57.1.

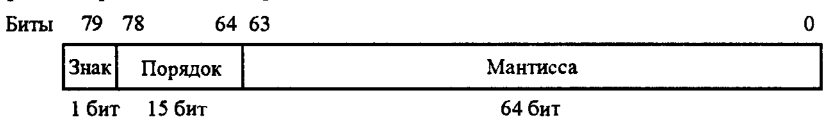


Рис. 57.1. Формат представления действительных чисел в сопроцессоре

Приведенный выше формат носит название формата расширенной точности. Заметим, что для представления отрицательных действительных чисел дополнительный код не используется. Знак числа в любом формате определяется старшим битом. Если он равен нулю, то число положительное, а если единице, то отрицательное. В расширенном формате под порядок отводится 15 бит и 64 бита под мантиссу.

Действительные числа могут быть представлены еще в двух форматах: одинарной точности (4 байта, из которых 23 бита отводятся для мантиссы, 8 – для порядка и 1 бит – для знака числа) и двойной точности (8 байт, 52 бита и 11 бит для мантиссы и порядка соответственно).

Какой из этих форматов используется, определяет содержимое 2-битового поля PC регистра управления сопроцессора (биты 8 и 9). PC=11 соответствует режиму расширенной точности, который автоматически устанавливается при инициализации сопроцессора. При работе в остальных двух режимах результаты вычислений округляются:

- если PC = 10, то выполняется округление до одинарной точности;
- если PC = 00, то выполняется округление до двойной точности.

Следует отметить, что ухудшение точности вычислений не приводит к ускорению работы программы. Поэтому эти режимы целесообразно использовать только в тех случаях, когда требуется выполнять расчеты с пониженной точностью: одинарной или двойной. Для перехода в них используется команда

```
fldcw  mem
```

(float load control word, загрузка слова управления сопроцессора), которая загружает в управляющий регистр слово из сегмента данных. Для записи содержимого управляющего регистра в память используется команда

```
fstcw  mem
```

(float store control word, сохранение слова управления сопроцессора).

Все действительные числа, независимо от используемого формата, хранятся в нормализованном виде. Нормализованным называется число, целая часть которого состоит из одной не равной нулю цифры. Поскольку при использовании двоичного представления эта цифра всегда равна единице, то она не хранится, что позволяет сэкономить один бит памяти. В поле порядка записывается степень числа 2, на которую умножается мантисса (плюс смещение, равное 16 383 для расширенной точности, 1023 для двойной точности или 127 для одинарной).

Рассмотрим, как представляется число 7,375 при использовании одинарной точности. Во-первых, заметим, что 7,375=4+2+1+1/4+1/8. Поэтому представление этого числа в двоичной форме имеет следующий вид: $111.011b=10^{10}*1.11011b$. Обратите внимание на то, что и основание системы счисления, и порядок, в который оно возводится, записаны в двоичной системе. Итак, хранимая мантисса числа будет равна 11011b, а порядок 2+127=129, в двоичной форме 10000001b. Поскольку число положительно, знаковый бит равен нулю.

Целиком двоичное нормализованное представление числа будет иметь вид, показанный на рис. 57.2.

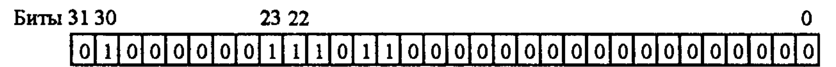


Рис. 57.2. Представление числа 7,375 в двоичной форме при использовании одинарной точности

Именно в таком виде и передаются числа из сопроцессора в память, если не выполнить приведения к целому формату (естественно, с округлением). Поэтому вывод действительных чисел не является тривиальной задачей. К счастью, содержимое регистров сопроцессора в отладчике можно посмотреть в обычной форме с плавающей точкой. Отлаживая программу, можно легко контролировать, как протекает процесс вычислений в сопроцессоре. Работе с отладчиком будет посвящена следующая статья, а сейчас еще раз напомним, что любая команда записи в стек сопроцессора выполняется аналогично команде push центрального процессора. Заметим, что стек невелик, его образуют всего 8 регистров, поэтому легко может произойти переполнение. Информация о содержимом регистров стека сведена в регистр признаков (тегов).

Регистр признаков состоит из восьми 2-битовых полей, которые обозначаются TAG0...TAG7. Каждое поле характеризует свой регистр стека (рис. 57.3).

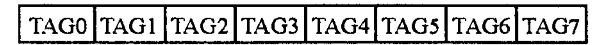


Рис. 57.3. Регистр признаков

По содержимому поля можно судить о том, какое число хранится в регистре. Если в поле признака находится 00, то в соответствующем регистре расположено действи-

тельное ненулевое число, 01 свидетельствует о наличии в регистре нуля, 11 означает, что регистр пуст, а 10 указывает на то, что он содержит недействительное число, например бесконечность.

Если регистр не отмечен как пустой, то при попытке записи в него вырабатывается код недействительной операции (устанавливается бит 0 регистра состояния) и запись в стек не производится.

Заметим, что этот особый случай может быть замаскирован, и тогда запись будет произведена. Для этого в бит 0 регистра управления надо записать 1, иначе возникновение особого случая вызовет прерывание центрального процессора.

Статья 58. **Отладка программ, работающих с сопроцессором**

Проверка правильности работы программы, включающей команды сопроцессора, затрудняется необходимостью использования процедур ввода и вывода, существенно более сложных, чем при работе с целыми числами. Поэтому контроль процесса выполнения подобных программ на практике производится только с помощью отладчиков, работа с одним из которых, TD, достаточно подробно рассмотрена в статье 4. Здесь мы коснемся только тех особенностей TD, которые относятся к отладке программ, использующих арифметику с плавающей точкой.

Прежде всего нам необходимо вывести на экран содержимое регистров сопроцессора. Для этого в меню пункта View необходимо выбрать пункт Numeric processor. В информационном кадре появится окно, в котором указаны имена и содержимое регистров стека сопроцессора и флагов. При отладке полезно также вывести окно регистров основного процессора. После этого надо определить такое положение окон, чтобы они не слишком загораживали друг друга и программу. Пример возможного расположения окон показан на рис. 58.1.

Теперь можно приступить к отладке программы. Проще всего работать в режиме пошаговой отладки. Для этого используем клавишу F8, каждое нажатие которой вызывает выполнение одного предложения программы. Изменения регистров, флагов и данных можно увидеть на экране. Отладчик позволяет не только последовательно, команда за командой выполнять программу, но и пошагово возвращаться назад (комбинация клавиш Alt+F4). Это возможно потому, что отладчик хранит последние выполненные команды. Для того чтобы вернуться к началу программы и выполнить ее еще раз, достаточно нажать Ctrl+F2.

В процессе отладки можно, не выходя из отладчика, изменять содержимое регистров и полей памяти, а также устанавливаемые флаги. Для этого используется локальное меню, открываемое нажатием комбинации клавиш Alt+F10 или нажатием правой кнопки мыши. Локальное меню контекстно чувствительно, т. е. для каждого активного окна имеется свое локальное меню.

```
■ File Edit View Run Breakpoints Data Options Window Help       READY
┌───Module: flt File: flt.asm 4──────────────────────────────┐┌───Regs-3───┐
  text segment 'code'                                          ax 0000  c=0
  assume cs:text,ds:data                                       bx 0000  z=0
  myproc  proc                                                 cx 0000  s=0
►         mov    ax,data  ;Инициализируем                      dx 0000  o=0
          mov    ds,ax    ;регистр DS                          si 0000  p=0
          fld    x        ;Загрузим первый операнд             di 0000  a=0
                          ;в стек сопроцессора                 bp 0000  i=1
          fld    y        ;Загрузим второй операнд             sp 0000  d=0
                          ;в стек сопроцессора                 ds 5D80
          fadd            ;Сложим их                           es 5D80
          fstp   z        ; ═[■]═80387 IPTR=2302E OPCODE=6D9 OPTR=00─4═[↑][↓]═
  outprog:                 Empty ST(0)                          im=1  ie=0
          mov    ax,4C00h  Empty ST(1)                          dm=1  de=0
          int    21h       Empty ST(2)                          zm=1  ze=0
  myproc  endp             Empty ST(3)                          om=1  oe=0
  text    ends             Empty ST(4)                          um=1  ue=0
  data segment             Empty ST(5)                          pm=1  pe=0
  x       dd     1.5       Empty ST(6)                         iem=0  ir=0
  y       dd     2.5       Empty ST(7)                          pc=3  cc=1
  z       dd     ?                                              rc=0  st=0
  data    ends                                                  ic=0
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Окно регистров

Окно сопроцессора

Рис. 58.1. Вывод содержимого регистров сопроцессора при работе с отладчиком TurboDebugger

Рассмотрим, как можно изменить содержимое регистров сопроцессора. Для этого в окне сопроцессора сначала выделим необходимый регистр, а затем вызовем локальное меню, которое включает три пункта: Zero, Empty и Change (рис. 58.2). Напомним, что при этом окно сопроцессора должно быть активным, поскольку локальное меню вызывается для активного окна.

```
■ File Edit View Run Breakpoints Data Options Window Help       MENU
┌───Module: flt File: flt.asm 10─────────────────────────────┐┌───Regs-3───┐
  text segment 'code'                                          ax 5D92  c=0
  assume cs:text,ds:data                                       bx 0000  z=0
  myproc  proc                                                 cx 0000  s=0
          mov    ax,data  ;Инициализируем                      dx 0000  o=0
          mov    ds,ax    ;регистр DS                          si 0000  p=0
          fld    x        ;Загрузим первый операнд             di 0000  a=0
                          ;в стек сопроцессора                 bp 0000  i=1
          fld    y        ;Загрузим второй операнд             sp 0000  d=0
                          ;в стек сопроцессора                 ds 5D92
►         fadd            ;Сложим их                           es 5D80
          fstp   z        ; ═[■]═80387 IPTR=5D90B OPCODE=106 OPTR=5D─4═[↑][↓]═
  outprog:                 Valid ST(0) 2.5                      im=1  ie=0
          mov    ax,4C00h  Valid ST(1) 1.5                      dm=1  de=0
          int    21h       Empty ST(2)       Zero               zm=1  ze=0
  myproc  endp             Empty ST(3)       Empty              om=1  oe=0
  text    ends             Empty ST(4)       Change...          um=1  ue=0
  data segment             Empty ST(5)                          pm=1  pe=0
  x       dd     1.5       Empty ST(6)                         iem=0  ir=0
  y       dd     2.5       Empty ST(7)                          pc=3  cc=1
  z       dd     ?                                              rc=0  st=6
  data    ends                                                  ic=0
Change the value of the highlighted register
```

Локальное меню

Рис. 58.2. Локальное меню окна сопроцессора

Если выбрать пункты Zero или Empty, то в регистр ST запишется нуль или же он освободится, т. е. станет доступным для записи. Для того чтобы записать в этот регистр новое число, надо выбрать пункт Change. Это вызовет появление окна, в поле которого надо ввести требуемое число и выбрать пункт OK. После этого содержимое регистра стека сопроцессора будет изменено. Аналогично можно изменить содержимое полей памяти и регистров основного процессора. Для переключения флага необ-

ходимо его выделить, вызвать локальное меню, в котором на этот раз будет только один пункт – Toggle, и подтвердить решение об изменении флага.

Чтобы при отладке следующей программы или в другом сеансе работы с той же программой не требовалось заново определять вид выводимой информации, можно запомнить текущую конфигурацию. Для этого выберите пункт меню Options, а в нем пункт Save Options. На экране появится окно, в котором необходимо определить сохраняемые параметры. Если нам необходимо сохранить выводимые окна, то в поле перед словом Layout должен быть установлен символ *. Если его нет, то он устанавливается нажатием пробельной клавиши. Переход на это поле осуществляется с помощью клавиши Tab. После завершения всех установок перейдите в поле OK и нажмите на Enter или выберите поле OK с помощью мыши. Теперь конфигурация запомнена. Для выхода из отладчика выберите в меню пункта File пункт Exit.

Заметим, что во всех приведенных выше примерах в процессе отладки мы работали с исходным текстом программы.

В ряде случаев бывает необходимо использовать деассемблированный текст. Для вывода его на экран в меню пункта View надо выбрать пункт CPU. Пример окна CPU с выведенным деассемблированным текстом для рассматриваемой программы показан на рис. 58.3.

Как видно из рисунка, окно, выводимое на экран при выборе пункта CPU, включает 5 внутренних окон. Левое верхнее окно содержит деассемблированный текст программы. Справа от него располагается окно регистров. Еще правее, в отдельном окне, выведены флаги процессора. В самом низу окна CPU располагается окно данных, в котором отображается дамп заданного участка памяти.

Последнее, пятое окно содержит дамп текущей вершины стека главного процессора. Оно располагается в правом нижнем углу окна CPU.

```
 ≡ File Edit View Run Breakpoints Data Options Window Help        READY
 [■]-CPU 80386                                                   5-[↑][↓]
#flt#myproc                                              ▲  ax 5D92  c=0
  cs:0000 B8925D          ♦ mov ax,data ;Инициализируем      bx 0000  z=0
  cs:0003 8ED8            ♦ mov ds,ax ;регистр DS            cx 0000  s=0
  cs:0005 9B               wait                              dx 0000  o=0
  cs:0006 D9060000         fld    dword ptr[#flt#x]          si 0000  p=0
  cs:000A 9B               wait                              di 0000  a=0
  cs:000B D9060400         fld    dword ptr[#flt#y]          bp 0000  i=1
  cs:000F►9B               wait                              sp 0000  d=0
  cs:0010 DEC1             faddp  st(1),st                   ds 5D92
  cs:0012 9B               wait                              es 5D80
  cs:0013 D91E0000         fstp   dword ptr[#flt#z]          ss 5D90
#flt#outprog                                                 cs 5D90
  cs:0017 B8004C          ♦ mov ax,4C00h                     ip 000F
  cs:001A CD21            ♦ int 21h
  cs:001C 0000             add    [bx+si],al             ▼
  ds:0000 00 00 C0 3F 00 00 20 40   └?  @                    ss:0000 0000
  ds:0008 00 00 00 00 00 00 00 00                            ss:0006 06D9
  ds:0010 FB 52 09 02 21 00 00 00 √R○●!                      ss:0004 9BD8
  ds:0018 07 00 1E 00 00 00 05 00 • ▲   ♣                    ss:0002 0E5D
  ds:0020 00 00 01 00 00 00 00 00   ☺                        ss:0000►92B8
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Рис. 58.3. Кадр отладчика с деассемблированным текстом программы

Статья 59. **Выполнение арифметических операций**

В распоряжении программиста, использующего сопроцессор, имеется около сотни команд. К ним относятся команды записи в стек сопроцессора целых и действитель-

ных чисел, сохранения полученных результатов в памяти, выполнения операций сложения, вычитания, умножения и деления, а также команды вычисления тригонометрических и некоторых других функций. Перечень команд приведен в Приложении 3; сейчас мы отметим их общие особенности.

Команды сопроцессора легко узнать, так как все они начинаются с буквы f (float). В этой статье для указания на произвольную команду сопроцессора будут использоваться обобщенные обозначения fcmd и fcmdp (второе – для команд с извлечением из стека).

Команды сопроцессора могут иметь два, один или ни одного явно задаваемого операнда. Однако фактически большинство команд являются двухоперандными; просто в ряде случаев эти операнды используются неявно.

В качестве операндов используются регистры сопроцессора и поля памяти. При указании поля памяти могут применяться все способы адресации, принятые в основном процессоре. Если, однако, один из операндов является полем памяти, то в качестве другого обязательно используется верхний регистр стека ST. В отличие от основного процессора, в командах сопроцессора не используются непосредственные операнды и, за исключением команды fstsw, не разрешается адресовать регистры основного процессора. Так же как и для основного процессора, оба операнда одновременно не могут быть полями памяти.

Команды обычно имеют операнд-источник и операнд-приемник. После выполнения команды операнд-источник не изменяется, а значение операнда-приемника замещается результатом. Если у команды определено два операнда, то первый является приемником, а второй источником. Возможные способы использования операндов показаны в табл. 59.1.

Таблица 59.1. Способы использования операндов в командах сопроцессора

Операнды	*Формат команды*	*Неявные операнды*	*Примеры*	*Результат*
Стек	fcmd	ST, ST(1)	fcom	ST-ST(1)
Память	fcmd mem	ST	fdiv mem	ST=ST/mem
Регистр	fcmd ST,ST(i)	нет	fsub ST, ST(4)	ST=ST-ST(4)
Регистр	fcmd ST(i),ST	нет	fmul ST(3), ST	ST(3)=ST(3)*ST
Регистр, съем со стека	fcmdp ST(i),ST	ST	fmulp ST(6), ST	ST(6)=ST(6)*ST, чтение
Регистр, съем со стека	fcmdp ST(i)	ST	faddp ST(5)	ST(5)=ST(5)+ST, чтение
Стек, съем со стека	fcmdp	ST, ST(1)	faddp	ST(1)=ST+ST(1), чтение

Рассмотрим простую программу, использующую сопроцессор. Приведенный в примере 59.1 фрагмент обеспечивает вычисление интеграла J от функции f методом Симпсона по трем точкам. В программе реализовано вычисление по формуле

```
J=h*(f(-h)+4f(0)+f(h))/3,
```

где h – шаг интегрирования.

Пример 59.1. Фрагмент программы вычисления интеграла

```
        xor    SI,SI        ;(1)Очистим регистр SI
        fld    f[SI]        ;(2)Загрузим f(-h)
        add    SI,4         ;(3)Увеличим смещение на 4 байта
```

```
        fld     f[SI]       ;(4)Загрузим f(0)
        add     SI,4        ;(5)Еще раз увеличим смещение
        fild    k4          ;(6)Загрузим коэффициент 4
        fmul                ;(7)Умножим 4 на f(0)
        fadd                ;(8)Получим f(-h)+4f(0)
        fld     f[SI]       ;(9)Загрузим f(h)
        fadd                ;(10)Получим f(-h)+4f(0)+f(h)
        fmul    h           ;(11)Умножим на шаг интегрирования
        fild    k3          ;(12)Загрузим коэффициент 3
        fdiv                ;(13)Получим h*(f(-h)+4f(0)+f(h))/3
        fadd    j           ;(14)Добавим к полученным ранее зна-
        fstp    j           ;(15)чениям и перенесем в поле j
;Поля данных
f       dd      2.7182818,1.9477340,1.6487213;Отсчеты функции
;Коэффициенты
k3      dw      3
k4      dw      4
h       dd      0.5
;Поле данных для интеграла
j       dd      ?
```

В этом фрагменте использован целый ряд способов адресации. Рассмотрим выполняемые в нем действия более подробно. После очистки регистра SI мы заносим в стек сопроцессора первый отсчет интегрируемой функции (предложение 2). Поскольку значения отсчетов заданы в сегменте данных, для этой цели используется команда загрузки стека, операндом которой является поле памяти. Для указания смещения использован косвенный метод адресации через индексный регистр SI.

Следующее предложение 3 является командой основного процессора. Она увеличивает смещение таким образом, чтобы в предложении 4 выполнилась загрузка очередного отсчета интегрируемой функции. Затем мы снова увеличиваем смещение и переходим к вычислению интеграла. Последний загруженный отсчет представляет собой f(0), поэтому мы загружаем коэффициент 4 (предложение 6) и умножаем его на f(0), используя команду умножения без явного указания операндов (предложение 7). Это возможно, так как к моменту выполнения предложения 7 в регистре ST(1) находится f(0), а в ST – коэффициент 4.

После того как с помощью предложений 8...13 значение интеграла вычислено, а при данных значениях отсчетов и шага оно будет равно 2.02632, мы суммируем его с ранее полученными результатами (предложение 14) и пересылаем в поле j сегмента данных (предложение 15). Для пересылки используем команду формата

```
        fcmdp mem
```

обеспечивающую удаление из стека пересылаемого значения *mem*.

В рассмотренном фрагменте встречаются как команды сопроцессора, так и команды основного процессора. Рассмотрим, например, предложения 1 и 2. В предложение 1 входит команда основного процессора, а в предложение 2 – команда сопроцессора; они выполняются друг за другом, и, кроме того, результат выполнения первой команды используется при выполнении второй. Поскольку сопроцессор и основной процессор могут работать параллельно, то для правильного выполнения этого фрагмента необходимо передать команду предложения 2 на выполнение сопроцессору только после того, как выполнится команда предложения 1. Как же это организуется при работе с сопроцессором? Обратим внимание на листинг программы 59.1, фрагмент которого приведен на рис. 59.1.

```
0000                text    segment
                    assume CS:text,DS:data;
0000                myproc proc
0000  B8 0000s      mov  AX,data
0003  8E D8         mov  DS,AX
0005  9B DB E3      finit
0008  33 F6         xor  SI,SI ;(1)Очистим регистр SI
000A  9B D9 84 0000r fld  f[SI] ;(2)Загрузим f(-h)
000F  83 C6 04      add  SI,4  ;(3)Увеличим смещение
0012  9B D9 84 0000r fld  f[SI] ;(4)Загрузим f(0)
0017  83 C6 04      add  SI,4  ;(5)Еще раз увеличим смещение
001A  9B DF 06 000Er fild k4    ;(6)Загрузим коэффициент 4
001F  9B DE C9      fmul       ;(7)Умножим 4 на f(0)
0022  9B DE C1      fadd       ;(8)Получим f(-h)+4f(0)
0025  9B D9 84 0000r fld  f[SI] ;(9)Загрузим f(h)
002A  9B DE C1      fadd       ;(10)Получим f(-h)+4f(0)+f(h)
002D  9B D8 0E 0010r fmul h     ;(11)Умножим на величину шага
0032  9B DF 06 000Cr fild k3    ;(12)Загрузим коэффициент 3
0037  9B DE F9      fdiv       ;(13)h*(f(h)+4f(0)+f(h))/3
003A  9B D8 06 0014r fadd j     ;(14)Добавим к полученным ранее
003F  9B D9 1E 0014r fstp j     ;(15)значениям и перенесем в j
0044  9B            fwait      ;(16)Подождем завершения команды
0045  B8 4C00       mov  AX,4C00h
0048  CD 21         int  21h
004A                myproc endp
004A                text ends
```

Рис. 59.1. Листинг программы примера 59.1

Из листинга видно, что код каждой из команд сопроцессора начинается с 16-ричного числа 9B. Это число является кодом команды wait, которая задерживает работу сопроцессора до тех пор, пока не закончит работу основной процессор. Таким образом, для обеспечения синхронизации ассемблер вставляет перед каждой командой сопроцессора команду wait. Это можно легко увидеть в отладчике, если пользоваться окном CPU, в которое выводится деассемблированный текст программы (рис. 59.2). Обратите, кстати, внимание на то, что в строках листинга, соответствующих командам сопроцесссора, после кода 9B стоит 16-ричная цифра D. Она является кодом первых 4 бит 5-битовой команды esc (полный код этой команды 11011), которая обеспечивает переключение на сопроцессор, и обязательно предшествует каждой его команде.

В поле деассемблированных команд окна CPU выводится полный адрес, код команды и ее мнемоническое обозначение. Перед каждой командой сопроцессора расположена строка с командой синхронизации wait. Обратите внимание, что команда fwait воспринята деассемблером отладчика как команда wait. Дело в том, что, хотя она начинается с буквы f и внешне выглядит как команда сопроцессора, ее код равен уже знакомому нам числу 9B (предложение 15 листинга на рис. 59.1) и, следовательно, совпадает с кодом команды wait основного процессора и является просто другим мнемоническим обозначением данной команды.

Команда fwait применяется в тех ситуациях, когда необходимо синхронизировать действия центрального процессора и сопроцессора, т. е. приостановить выборку команд из очереди до завершения сопроцессором текущей команды. Ее наличие приводит к тому, что центральный процессор не может обратится к операнду в памяти до тех пор, пока этот операнд не будет записан в память сопроцессором. Пример 59.2 иллюстрирует подобную ситуацию.

```
■ File Edit View Run Breakpoints Data Options Window Help          READY
[■]-CPU 80386--------------------------------------------------3-[↑][↓]-
 cs:002A▶9B              wait                       ▲  ax 5D95   c=0
 cs:002B DEC1            faddp   st(1),st              bx 0000   z=0
 cs:002D 9B              wait                          cx 0000   s=0
 cs:002E D80E1000        fmul    dword ptr[#ch81#h]    dx 0000   o=0
 cs:0032 9B              wait                          si 0008   p=0
 cs:0033 DF060C00        fild    word ptr[#ch81#k3]    di 0000   a=0
 cs:0037 9B              wait                          bp 0000   i=1
 cs:0038 DEF9            fdivp   st(1),st              sp 0000   d=0
 cs:003A 9B              wait                          ds 5D95
 cs:003B D8061400        fadd    dword ptr[#ch81#j]    es 5D80
 cs:003F 9B              wait                          ss 5D90
 cs:0040 D91E1400        fstp    dword ptr[#ch81#j]    cs 5D90
 cs:0044 9B              wait                          ip 002A
 cs:0045 B8004C          mov     ax,4C00
 cs:0048 CD21            int     21                 ▼
 es:0000 CD 20 FF 9F 00 9A F0 FE                       ss:0008 F633
 es:0008 56 C9 E8 01 72 22 AA 01                       ss:0006 E3DB
 es:0010 72 22 89 02 CD 1C 3F 0E                       ss:0004 9BD8
 es:0018 01 01 01 00 07 FF FF FF                       ss:0002 8E5D
 es:0020 FF FF FF FF FF FF FF FF                       ss:0000▶95B8
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Рис. 59.2. Кадр отладчика с деассемблированным текстом программы

Пример 59.2. Использование команды fwait

```
        fstsw  sword       ;Слово состояния -> в поле sword
        fwait              ;Ждем, пока сопроцессор не завершит
                           ;выполнения команды fstsw
        mov    AX,sword    ;Слово состояния -> в регистр AX
;Поля данных
sword   dw     ?
```

Подобные фрагменты мы будем использовать в последующих статьях в тех случаях, когда потребуется передать в основной процессор значения флагов слова состояния сопроцессора.

Статья 60. **Использование сопроцессора для реализации операции возведения положительного числа в дробную степень**

Наличие сопроцессора позволяет существенно упростить решение задач, использующих показательные функции. Это происходит из-за того, что такие функции вычисляются с помощью одной или нескольких команд и не требуется составлять специальные относительно сложные процедуры. Рассмотрим использование сопроцессора для вычисления функции y=s*sqrt(x) и вывода ее графика на экран. Вычислим 400 значений функции и выведем их на экран в виде красных точек. Для этого прежде всего с помощью функции 0h прерывания 10h BIOS установим графический режим 10h (VGA с разрешением 640*350, 16 цветов). Затем перешлем в стек сопроцессора значение коэффициента s и вычислим значение функции. Для вывода точек на экран воспользуемся функцией 0Ch прерывания 10h BIOS. Фрагмент программы приведен в примере 60.1.

*Пример 60.1. Вычисление и вывод на экран графика функции y=a*sqrt(x)*

```
;Установка параметров
        mov    AH,0        ;Функция установки графического режима
        mov    AL,10h      ;Режим 10h-графика, 640*350 точек
```

```
        int     10h             ;Прерывание BIOS
        fild    s               ;Зашлем коэффициент s в регистр ST
        mov     CX,400          ;Число повторений цикла
;Цикл вычисления координат и вывода точек на экран
line:   push    CX              ;Сохраним счетчик цикла
        fild    x               ;Зашлем x в ST, s идет в ST(1)
        fsqrt                   ;Корень из ST, результат там же
        fmul    ST,ST(1)        ;Умножим на s, результат в ST
        fistp   y               ;Перешлем содержимое ST в ячейку y
        mov     AH,0Ch          ;Функция вывода пиксела
        mov     AL,4            ;Цвет пиксела
        mov     BH,0            ;Видеостраница 0
        mov     CX,x            ;Номер столбца
        mov     DX,100
        sub     DX,y            ;Номер строки
        int     10h             ;Прерывание BIOS
        inc     x               ;Сместимся вправо
        pop     CX              ;Восстановим счетчик цикла
        loop    line            ;Повторим CX раз
;Поля данных
forcolor db     4               ;Цвет
x       dw      0
y       dw      ?
s       dw      4
```

Непосредственно для работы с сопроцессором здесь использовано всего 5 команд. С их помощью производится засылка в стек сопроцессора коэффициента s и текущего аргумента x, вычисление sqrt(x) с сохранением результата в стеке, вычисление произведения s*sqrt(x) и пересылка полученного результата в ячейку y. Применение команды fist позволяет перед пересылкой привести число к целому формату. Поэтому его сразу можно использовать для указания координаты y.

Вычисление произвольной степени положительного числа представляет собой более сложную задачу. Для ее решения можно использовать команды сопроцессора f2xm1 (float 2^x-1, вычисление в формате плавающей точки выражения 2^x-1) и fyl2x (float $y*\log_2 x$, вычисление в формате плавающей точки выражения $y*\log_2 x$). Команда f2xm1 вычисляет 2 в степени ST минус 1, при этом значение ST должно находиться в диапазоне от 0.0 до 0.5. Команда fyl2x производит умножение содержимого регистра ST(1) на логарифм ST по основанию 2. Таким образом, чтобы возвести число в произвольную степень, имея в наличии эти две команды, необходимо вычислить логарифм (по основанию 2) от основания степени, умножить его на показатель степени и возвести 2 в степень, значение которой равно получившемуся результату. Если при этом соблюдается условие для выполнения команды f2xm1, то процедура будет очень простой. Рассмотрим, например, как вычислить кубический корень из конкретного числа (пример 60.2).

Пример 60.2. Вычисление кубического корня из числа 2,5

```
        fld1                    ;Загрузим в стек сопроцессора 1
        fld     n               ;Загрузим в стек степень корня
        fdiv                    ;Вычислим показатель степени,
                                ;в которую будет возводиться число
        fld     a               ;Загрузим основание
        fyl2x                   ;Вычислим логарифм искомого числа
        f2xm1                   ;Вычислим (корень-1)
        fld1                    ;Загрузим единицу
        fadd                    ;Получаем корень
        fstp    rez             ;Пересылаем результат в память
```

```
;Поля данных
a       dd   2.5
n       dd   3.0
rez     dd   ?
```

В приведенном примере команда fdiv (float divide, деление чисел в формате плавающей точки) использована для того, чтобы в сегменте данных задавать показатель корня, а не дробную степень. В результате ее выполнения в регистр ST засылается ST(1)/ST. Регистр ST освобождается.

Если показатель степени, в которую надо возвести число 2, чтобы получить результат, представляет собой положительное число, то перед выполнением команды f2xm1 надо выделить его целую часть. Затем следует определить диапазон, в котором лежит дробная часть. Если дробная часть превышает 0.5, то надо уменьшить ее на эту величину и домножить результат на квадратный корень. И только после этого можно выполнять команду f2xm1. Пример 60.3 содержит фрагмент программы, включающий операции проверки.

Пример 60.3. Фрагмент программы возведения положительного числа в произвольную положительную степень

```
        finit
        fld   n              ;Загрузим в стек показатель степени
        fld   a              ;Загрузим основание
        fyl2x                ;Вычислим логарифм по основанию 2
        fst   ST(1)          ;Запишем его в регистр ST(1)
;Установим способ округления
        fstcw contrl         ;Слово управления сопроцессора → contrl
        fwait                ;Подождем выполнения этой команды
        or    contrl,0C00h;Режим отбрасывания дробной части
        fldcw contrl         ;Загрузим новое слово управления
        frndint              ;Округляем, получая целую часть
;Если целая часть не равна нулю, то передадим полученное число в
;основной процессор и вычислим 2 в степени, равной целой части
        fist  stn
        cmp   stn,0
        jz    cont
        fsub                 ;Получим дробную часть
        fwait
;На основном процессоре вычислим 2 в степени, равной целой части
        mov   CX,stn
        sub   CX,1
        mov   AX,2
        shl   AX,CL
        mov   res,AX
        fild  res            ;Загрузим результат в стек
        fstp  rezlt          ;и в память в виде действительного числа
;Проверим величину получившегося остатка
cont:   fst   ST(1)          ;Сохраним получившийся остаток
        fild  const2         ;Загрузим 2 в стек сопроцессора
        fmul                 ;Помножим остаток на 2
        frndint              ;и отбросим дробную часть
        fistp pj             ;И в память в виде дробного числа
        fwait                ;Ожидание выполнения операции
        cmp   pj,1           ;Сравним с 1
;Если остаток меньше 0.5, то перейдем к вычислению 2 в степени,
;равной остатку, иначе - домножим результат на корень из двух
        jl    cont1
        fsub  const05
        fild  const2
```

```
        fsqrt
        fmul   rezlt        ;Домножим результат на корень из двух
        fstp   rezlt        ;Запишем в ячейку rezlt
contl:  f2xml               ;Вычислим 2 в степени, равной остатку
        fldl                ;Выполним коррекцию
        fadd
        fld    rezlt
        fmul                ;Получим искомое число,
        fstp   rezlt        ;которое перешлем в память
;Поля данных
a       dd     125.0        ;Число, возводимое в степень
n       dd     0.33333333   ;Показатель степени
rezlt   dd     ?            ;Поле для сохранения результата
const2  dw     2            ;Используемые константы
const05 dd     0.5
res     dw     0            ;Поля для хранения промежуточных данных
stn     dw     ?
pj      dw     ?
contrl  dw     ?            ;Поле для сохранения слова
                            ;управления сопроцессора
```

Приведенную программу можно оптимизировать, выполняя сравнение в сопроцессоре и округляя получившийся после вычисления логарифма результат до ближайшего целого. Затем следует возвести 2 в это целое число и, в зависимости от знака разности между округленным и неокругленным числами, выбрать один из двух вариантов. Можно также умножить 2 в степени, равной целой части, на 2 в степени, равной остатку, или разделить на него.

Статья 61. **Вычисление корня нелинейного уравнения F(x)=0**

Удобство использования сопроцессора легко иллюстрируется на примере решения задачи определения корня уравнения F(x)=0 с заданной точностью delta<<1. Широко известный метод половинного деления состоит из следующих операций. Сначала вычисляются значения функции в точках, расположенных через равные интервалы на оси x. Это делается до тех пор, пока не будут найдены два последовательных значения F(x[n]) и F(x[n+1]), имеющие противоположные знаки. Если функция непрерывна, то изменение знака указывает на существование корня. Предположим, что x[n] и x[n+1] уже найдены и x[n]=a, x[n+1]=b, причем F(a)<0, а F(b)>0. Кроме того, на этом отрезке функция Y=F(x) имеет один 0. Тогда график функции пересекает ось только в одной точке z, которую и требуется найти.

Для этого вычисляется точка c=(a+b)/2. Если F(c)>0, то заменяем точку b на точку c, иначе точкой c заменяется точка a. Процедуру продолжаем до тех пор, пока не выполнится условие |F(a)-F(b)|<delta. После этого z определяется как (a+b)/2. Эффективное решение такой задачи возможно только при наличии сопроцессора. Фрагмент программы, реализующей поиск корня уравнения x*x–p*sqrt(x)=0 на интервале [0.0001,1], приведен в примере 61.1.

Пример 61.1. Программа вычисления корня уравнения F(x)=0

```
cont:                   ;Начало цикла
;Вычисление F(a)
        fld    a        ;a → ST
```

```
        fsqrt               ;sqrt(x) → ST
        fld     p           ;sqrt(x) → ST(1), p → ST
        fmul                ;p*sqrt(x) → ST
        fld     a           ;p*sqrt(x) → ST(1), a → ST
        fld     a           ;p*sqrt(x) → ST(2), a → ST(1), a → ST
        fmul                ;a*a → ST(0), p*sqrt(x) → ST(1)
        fadd                ;F(a) → ST
;Вычисление F(b)
        fld     b           ;F(a) → ST(1), b → ST
        fsqrt
        fld     p
        fmul
        fld     b
        fld     b
        fmul
        fadd                ;F(b) → ST, F(a) → ST(1)
;F(a) и F(b) вычислены
        fsub                ;F(a)-F(b) → ST
        fabs                ;¦F(a)-F(b)¦ → ST
        fcom    delta       ;Проверка результата (¦F(a)-F(b)¦<delta)
        fstsw   sw          ;Слово состояния в основной процессор
        fwait
        mov     AX,sw
        and     AX,0500h    ;выделение битов C2 и C0
        cmp     AH,1        ;Проверка наличия комбинации C2=0, C0=1
        je      ed          ;z найдено!
        ffree   ST(0)       ;Очистка ST
        fld     a           ;вычисление z
        fld     b
        fadd
        fdiv    dlt         ;Деление на 2, z → ST
        fst     z           ;Результат в z
;Вычисление F(z)
        fsqrt
        fld     p
        fmul
        fld     z
        fld     z
        fmul
        fadd                ;F(z) → ST
        ftst                ;F(z)>0?
        fstsw   sw          ;Слово состояния в основной процессор
        fwait
        mov     AX,sw
        and     AX,8600h    ;выделение битов C3, C2 и C1
        cmp     AH,0        ;Проверка наличия C3=0, C2=0, C1=0
        je      cont1
        ffree   ST(0)       ;Очистка ST
        fld     z
        fstp    b
        jmp     cont        ;z пока не найдено
cont1:  ffree   ST(0)
        fld     z
        fstp    a
        jmp     cont        ;z пока не найдено
;Вычисление z
ed:     fld     a
        fld     b
```

```
        fadd
        fdiv    dlt
        fst     z
;Поля данных
a       dd      0.0001
b       dd      1.0
z       dd      (?)
```

Сперва инициализируем сопроцессор с помощью команды finit. Затем последовательно вычислим значения F(a) и F(b). Для этого сначала перешлем из памяти значение переменной a в регистр ST и вычислим квадратный корень. Результат останется в ST. После этого в стек сопроцессора зашлем значение коэффициента p. Оно будет записано в вершину стека ST. К регистру, в котором находится уже вычисленный корень, мы теперь будем обращаться как к ячейке стека ST(1). Затем с помощью команды fmul (float multiply, умножение чисел в формате плавающей точки) без параметров перемножим содержимое регистров ST и ST(1). Значение регистра ST(1) после выполнения этой операции становится неопределенным, а произведение засылается в ST. После этого два раза засылаем величину a в стек. С помощью команды умножения вычислим квадрат a, который также запишем в стек. Далее с помощью команды сложения вычислим значение функции F(a).

Для вычисления F(b) используем такую же последовательность операций. Весь процесс обработки происходит аналогично, за исключением того, что в стеке сохраняется значение F(a).

После того как F(a) и F(b) вычислены, определяем их разность с помощью команды fsub (float substract, вычитание в формате плавающей точки) без явного указания операндов. Эта команда вычисляет ST(1)-ST, записывает результат в ST(1) и выполняет считывание из стека. Так как корень считается найденным, если модуль вычисленной разности меньше заданной величины delta, предварительно с помощью команды fabs (float absolute, абсолютное значение числа в формате плавающей точки) находим модуль ST. Для сравнения используем команду fcom (float compare, сравнение чисел в формате плавающей точки), которая из содержимого ST вычитает операнд, в данном случае delta. Результат операции не сохраняется, происходит лишь установка битов C0 и C2 в регистре состояния сопроцессора.

Для того чтобы выполнить переход по результатам сравнения, необходимо использовать команды основного процессора, поскольку сопроцессор таких команд не имеет. Следовательно, необходимо передать результаты сравнения из сопроцессора в основной процессор. Для этого с помощью команды

```
        fstsw  sw
```

передадим содержимое регистра состояния в ячейку памяти sw, а затем проанализируем значения битов C0 и C2. Перед выполнением команды

```
        mov    AX,sw
```

используем команду fwait для синхронизации работы основного процессора и сопроцессора. После того как регистр флагов центрального процессора установлен, проверим, найден ли диапазон, содержащий корень. И если этот диапазон найден, то переходим по метке ed на фрагмент, вычисляющий корень. Точность полученного решения будет равна |F(z)|.

В противоположном случае вычисляем точку z и выполняем переприсваивание: если знак F(z) совпадает со знаком F(a), то a заменяем на z, иначе на z заменяем b. Как

и ранее, после проверки знака для выполнения перехода передаем в сопроцессор содержимое слова состояния сопроцессора.

Статья 62. **Процедура рисования окружности**

Без использования сопроцессора практически невозможно обойтись в тех областях, где применяются процедуры машинной графики. Особенно это важно при разработке программ, реализующих режимы мультипликации и анимации. Если вычисление тригонометрических функций реализовывать на основном процессоре, то программа будет выполняться недопустимо медленно даже при выводе сравнительно несложных изображений.

Рассмотрим программу, которая обеспечивает вывод на экран окружности заданного радиуса. Координаты точек окружности будем вычислять с шагом в 1 градус. Таким образом, нам понадобится реализовать цикл по независимой переменной от 0 до 360 градусов. Заметим, что перед вычислением аргумент должен быть преобразован в радианы. Текст программы приведен в примере 62.1.

Пример 62.1. Вычисление координат точек окружности и вывод их на экран

```
.386
data     segment
;Поля данных
x360     dd     180.0        ;Константа перевода градусы-радианы
x36      dw     360          ;Число точек на окружности
forcolor db     10           ;Салатовый цвет
;Координаты центра окружности
xc       dw     320
yc       dw     175
;Значения радиуса по осям
rx       dw     100
ry       dw     70
;Переменные
x        dw     ?            ;Текущие координаты точки окружности
y        dw     ?
angl     dw     1            ;Текущее значение угла
data     ends
text     segment use16
         assume CS:text,DS:data
;Подпрограмма изменения цвета пиксела
point    proc
         push   CX
         mov    CX,xc
         mov    AH,0Ch
         mov    AL,forcolor
         mov    BH,0
         fld    yc1
         fistp  yc
         mov    DX,yc
         fld    xc1
         fistp  xc
         mov    CX,xc
         sub    CX,x
         sub    DX,y
         int    10h
         pop    CX
         ret
point    endp
```

```
;Главная процедура
main    proc
;Подготовка данных
        mov     AX,data         ;Инициализация
        mov     DS,AX           ;регистра DS
        mov     AH,0h           ;Установка графического режима
        mov     AL,10h          ;Режим 10h
        int     10h             ;Прерывание BIOS
        mov     CX,x36          ;Число шагов построения окружности
        finit                   ;Инициализация сопроцессора
        fldpi                   ;Загрузка в стек числа pi
        fld     x360            ;Загрузка в стек числа 360,
        fdiv                    ;pi/360, результат в ST
        fstp    x360            ;Сохранение в памяти коэффициента
                                ;перевода градусов в радианы
;вычисление координат точек и вывод рисунка
do:     fld     x360            ;Коэффициент градус-радианы в стек
        fild    angl            ;Очередное значения угла в стек
        fmul                    ;Перевод в радианы
        fsincos                 ;sin(x) -> ST(1), cos(x) -> ST(0)
        fild    ry              ;Загрузка радиуса по координате y
        fmul                    ;Вычисление координаты y
        fistp   y               ;Запись ее в память в формате целого
                                ;числа с извлечением из стека
        fild    rx              ;Загрузка радиуса по координате x
        fmul                    ;Вычисление координаты x
        fistp   x               ;Запись ее в память в формате целого
                                ;числа с извлечением из стека
        fwait                   ;Ожидание завершения работы сопроцессора
        call    point           ;Вывод точки на экран
        inc     angl            ;Приращение угла
        loop    do              ;Цикл
;Задержка до нажатия клавиши
        mov     AH,8
        int     21h
        mov     AX,4C00h        ;Выход в DOS с кодом ошибки 0
        int     21h
main    endp
text    ends
stk     segment stack 'stack'
        dw      128 dup(?)
stk     ends
end     main
```

Программа начинается с директивы .386, которая разрешает компиляцию всех команд процессоров 80386/87. При этом сегмент команд необходимо объявить с описателем usc16.

Перед выполнением вычислений мы устанавливаем графический режим с помощью функции 0h прерывания 10h BIOS, инициализируем сопроцессор и вычисляем коэффициент для пересчета угла из градусов в радианы. Дело в том, что аргументы команд, вычисляющих тригонометрические функции, должны быть заданы в радианах, а приращение углов удобнее задавать в градусах. Для загрузки в стек числа pi можно использовать команду fldpi (float load pi, загрузка числа pi), одну из набора специальных команд, загружающих константы. После того как число pi загружено в стек, загружаем из памяти эквивалентное значение в градусах и с помощью команды деления вычисляем коэффициент.

Вся остальная часть программы практически состоит из одного цикла, в котором производится вычисление значений синуса и косинуса, определение координат точки окружности и изменение цвета пиксела с указанными координатами на экране. Заметим, что значение радиуса по горизонтальной оси отличается от значения радиуса по вертикальной. Это происходит из-за того, что в графических режимах разрешения по этим осям различны.

Полученную программу легко преобразовать для вывода других геометрических фигур, например спирали. В этом случае потребуется ввести в ее текст дополнительную подпрограмму, обеспечивающую непрерывное изменение радиуса. В результате у нас получится следующая программа (пример 62.2).

Пример 62.2. Фрагменты программы вычисления и вывода на экран спирали

```
.386
;Сегмент данных
        ...
;Дополнительно введенные переменные
del      dd     0.9985       ;Коэффициент сжатия спирали
delx     dd     1.0000004;Коэффициент перемещения по оси x
dely     dd     0.999997     ;Коэффициент перемещения по оси y
forcolor db     9            ;Голубой цвет
rx       dd     100.0        ;У этих переменных изменен тип, не
ry       dd     70.0         ;забудьте изменить соответствующие команды
xc1      dd     320.0        ;Дополнительные переменные для
yc1      dd     175.0        ;преобразования координат центра
;Сегмент команд
text     segment use16
         assume CS:text,DS:data
;Подпрограмма изменения координат центра и радиуса
coord    proc
         fld    rx
         fmul   del
         fstp   rx
         fld    ry
         fmul   del
         fstp   ry
         fild   xc1
         fmul   delx
         fistp  xc1
         fild   yc1
         fmul   dely
         fistp  yc1
         ret
coord    endp
;Подпрограмма изменения цвета пиксела (см. пример 62.1)
point    proc
         ...
point    endp
;Главная процедуры
main     proc
;Подготовка данных
         ...
;вывод точки на экран
         call   point
         inc    angl
;Изменение координат и радиуса
         call   coord
         loop   Do
;Задержка до нажатия клавиши
```

```
        mov     AH,8
        int     21h
        mov     AX,4C00h    ;выход в DOS с кодом ошибки 0
        int     21h
main    endp
text    ends
stk     segment stack 'stack'
dw      128 dup (?)
stk     ends
end     main
```

В этой программе использована дополнительная подпрограмма coord, обеспечивающая непрерывное уменьшение размера радиуса и изменение координат центра окружности. Каждая из указанных выше величин записывается в стек сопроцессора. Затем она умножается на действительный коэффициент, после чего полученное значение снова записывается в память и выполняется чтение из стека для того, чтобы избежать его переполнения.

Попробуйте самостоятельно составить программу, использующую вычисление тригонометрических функций, например выводящую на экран график функции Y = = Sin(a*x+b). Для этого вам надо будет воспользоваться командой fsin.

Статья 63. **Управляющие регистры сопроцессора**

Рассмотрим группу регистров, обеспечивающих управление работой сопроцессора. Эти регистры часто называют управляющими или нечисловыми регистрами; к ним относятся: управляющий регистр, регистр состояния, регистр признаков, указатель команды и указатель операнда (рис. 63.1).

Для работы с этими регистрами в сопроцессоре используется 30 команд, которые могут начинаться либо с двух букв fn, либо с одной буквы f, например: fnstcw и fstcw. Если такая команда начинается с буквы f (float, обработка чисел с плавающей точкой), то перед тем, как передать ее на выполнение сопроцессору, центральный процессор проверяет, занят ли сопроцессор. Для этого проверяется бит занятости (B) регистра состояния, а также биты ошибок. Если команда начинается с букв fn (float no wait, обработка чисел с плавающей точкой без ожидания), то она передается на выполнение сопроцессору без каких-либо предварительных проверок.



Рис. 63.1. Нечисловые регистры сопроцессора

С одной из команд работы с нечисловыми регистрами, именно с командой инициализации сопроцессора finit, мы уже знакомы. Она устанавливает начальные значения в регистре состояния, управляющем регистре и регистре признаков. Регистр признаков, содержащий сведения о данных, находящихся в каждом из числовых регистров сопроцессора, был уже рассмотрен в статье 57. Команда finit засылает во все поля этого регистра значения 11b. Это означает, что все числовые регистры сопроцессора свободны и в них можно записывать данные. Таким образом, команда finit может использоваться для очистки сразу всех числовых регистров.

Если же требуется очистить определенный числовой регистр, например ST(i), то надо использовать команду

```
ffree ST(i)        ;float free, освобождение регистра
```

Эта команда записывает в i-е поле регистра признаков 11b, тем самым помечая числовой регистр ST(i) как свободный.

Содержимое регистра состояния показано на рис. 63.2.



Рис. 63.2. Структура регистра состояния сопроцессора

Биты с 0 по 5 представляют собой флаги особых случаев. Они устанавливаются при возникновении следующих ошибок:

- IE (Invalid Operation) – недействительная операция;
- DE (Denormalized Operand) – денормализованный операнд;
- ZE (Zero Divide) – деление на нуль;
- OE (Overflow) – переполнение;
- UE (Underflow) – антипереполнение;
- PE (Precision) – потеря точности.

Бит 6 содержит флаг стека SF (Stack Flag), а бит 7 – слова состояния – флаг суммарной ошибки ES (Summary Error), который устанавливается при возникновении незамаскированного особого случая.

Биты C0, C1, C2 и C3 (Condition Code) – коды условий. Они определяются по результату выполнения команд сравнения и команд нахождения остатка.

В поле ST (Stack Top Pointer) содержится номер числового регистра, являющегося вершиной стека. Бит занятости B (Busy) устанавливается, если сопроцессор выполняет команду или происходит прерывание от основного процессора. Если сопроцессор свободен, то бит занятости сбрасывается.

При инициализации сопроцессора все флаги, за исключением ST и ES, значения которых не определяются, сбрасываются.

Для того чтобы получить информацию о содержимом полей регистра состояния или изменить их, используются следующие команды:

fstsw (float store state word) и fnstsw mem – записать слово состояния сопроцессора в память;

fstsw AX и fnstsw AX – записать слово состояния сопроцессора в регистр AX (только для сопроцессоров 80287+);

fclex (float cleare exceptions) и fnclex – сбросить все флаги ошибок, а также биты ES и B;

fincstp (float increment stack pointer, увеличить указатель стека с плавающей точкой) – увеличить указатель вершины стека числовых регистров на единицу;

fdecstp (float decrement stack pointer, уменьшить указатель стека с плавающей точкой) – уменьшить указатель вершины стека числовых регистров на единицу.

Напомним, что, хотя содержимое полей кодов условий устанавливается при выполнении команд сопроцессора, выполнить переход можно только путем анализа установленного кода в основном процессоре. Для этого надо предварительно переписать

в ячейку памяти содержимое слова состояния. Таким образом, если условием завершения итераций является выполнение неравенства |p| <eps, где p – текущий параметр, а eps – заданная величина, то проверку можно выполнить так, как указано в примере 63.1.

Пример 63.1. Проверка условия выхода из цикла

```
...
fld    p             ;Поместим p в стек сопроцессора
fabs                 ;Вычислим модуль p
fcom   eps           ;Сравним с eps, результат в битах C0…C2
                     ;регистра состояния
fstsw  AX            ;Запишем регистр состояния в AX
and    AX,0700h      ;Выделим биты C0…C2
cmp    AH,1h         ;Если p<eps,то только C0=1. Проверим это
je     output        ;Если это так, выйдем из цикла
jmp    cont          ;Иначе продолжим вычисления
...
```

Структура регистра управления, отдельные поля которого были нами рассмотрены ранее, показана на рис. 63.3.

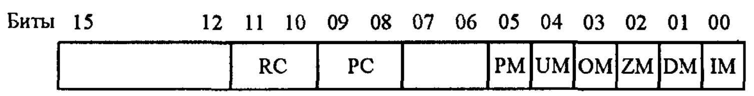


Рис. 63.3. Структура регистра управления сопроцессора

Биты 0...5 являются масками недействительных состояний. Если бит маски для какого-либо недействительного состояния, например переполнения, сброшен, то возникновение этого состояния вызывает прерывание центрального процессора. Если этот бит установлен, то прерывание не вырабатывается, а в качестве результата формируется особое значение (в рассматриваемом случае – код бесконечности). Используются следующие маски особых случаев:

- IM (Invalid Operation Mask) – маска недействительной операции;
- DM (Denormalized Operand Mask) – маска денормализованного результата;
- ZM (Zero Divide Mask) – маска деления на нуль;
- OM (Overflow Mask) – маска переполнения;
- UM (Underflow Mask) – маска антипереполнения;
- PM (Precision Mask) – маска особого случая при неточном результате.

Содержимое поля PC (Precision Control, биты 8 и 9) определяет точность вычислений в сопроцессоре:

- 11b – используется расширенная точность;
- 10b – результат округляется до двойной точности;
- 00b – результат округляется до одинарной точности.

Двухбайтовое поле RC (Rounding Control, биты 10 и 11) определяет режим округления при выполнении операций с вещественными числами:

- 00b – производится округление к ближайшему числу. Этот режим устанавливается при инициализации сопроцессора;
- 01b – производится округление в направлении к отрицательной бесконечности;
- 10b – производится округление в направлении к положительной бесконечности;
- 11b – производится округление в направлении к нулю.

Бит 12 предназначен для управления трактовкой понятия бесконечности; этот бит называется IC (Infinity Control). Данные сопроцессоры могут работать в двух режимах: проективном (IC=0) и аффинном (IC=1). В проективном режиме существует только одна бесконечность, которая не имеет знака. В аффинном режиме определено две бесконечности: положительная и отрицательная. В этом режиме допускается выполнение арифметических операций с бесконечностями. Современные сопроцессоры работают только в режиме аффинной арифметики.

После выполнения команды finit в регистре управления устанавливается режим работы с расширенной точностью и округления к ближайшему числу. Все биты масок обработки особых случаев устанавливаются в 1; следовательно, все особые случаи будут замаскированы.

Для работы с регистром управления используются следующие команды сопроцессора:

fstcw (fnstcw) mem – записать содержимое управляющего регистра сопроцессора в ячейку памяти mem;

fldcw mem – записать содержимое ячейки памяти mem в управляющий регистр сопроцессора.

Рассмотрим, каким образом можно определить и при необходимости изменить установленный режим округления. Фрагмент программы, выводящей на экран информацию о режиме округления, приведен в примере 63.2.

Пример 63.2. Фрагмент программы, анализирующий содержимое поля RC регистра управления сопроцессора

```
.386
data segment
;Выводимые сообщения
mes00   db    'Округление до ближайшего целого$'
mes01   db    'Округление в сторону уменьшения$'
mes10   db    'Округление в сторону увеличения$'
mes11   db    'Отбрасывание разрядов$'
;Резервируем слово для хранения регистра управления
clwd    dw    ?
        ...
data    ends
text    segment use16
assume  CS:text,DS:data
;Подпрограмма вывода сообщения
outt    proc
        mov   AH,09h
        int   21h
        ret
outt    endp
;Основная программа
main    proc
        mov   AX,data
        mov   DS,AX
        ...
;Проверка установленного режима округления
        fstcw clwd        ;Запись регистра управления в ячейку clwd
        mov   AX,clwd     ;Перенесем содержимое clwd в AX
        and   AX,0C00h    ;Выделим биты, управляющие режимом
        shr   AH,2        ;округления и сдвинем AH вправо на 2 бита
;Определяем, какой режим, и выводим соответствующее сообщение
        cmp   ah,0h
```

```
        jg      m01
        mov     DX,offset mes00
        jmp     ot
m01:    cmp     ah,1h
        jg      m02
        mov     DX,offset mes01
        jmp     ot
m02:    cmp     ah,2h
        jg      m03
        mov     DX,offset mes10
        jmp     ot
m03:    mov     DX,offset mes11
ot:     call    outt
        ...
main    endp
text    ends
end     main
```

Для того чтобы установить необходимый режим округления, достаточно считать текущее слово управления, изменить необходимым образом содержимое поля RC и затем записать новое слово в регистр управления.

С помощью отладчика можно просматривать и изменять флаги регистра состояния и маски регистра управления непосредственно в процессе отладки программы. Вся эта информация, пример которой приведен на рис. 63.4, выводится в правой части окна сопроцессора.

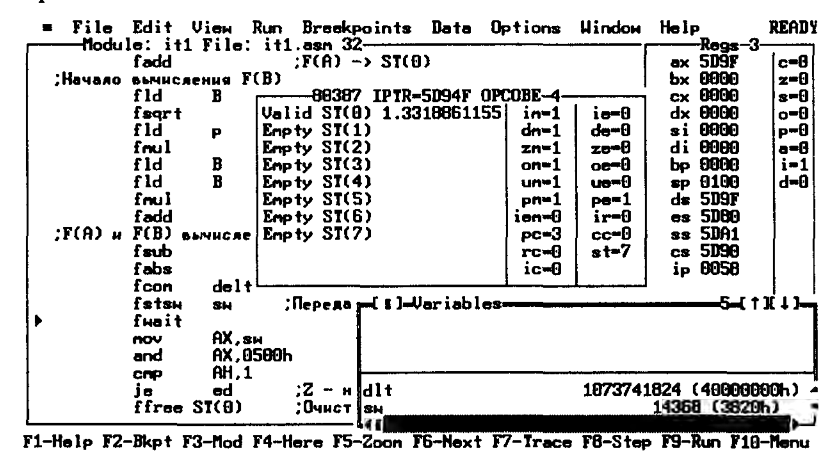


Рис. 63.4. Вывод информации о содержимом регистров состояния и управления

Приведенные здесь флаги и поля битов нам уже знакомы, отметим только, что содержимое всех кодов условий представлено одним параметром CC (Condition Code). Для изменения флагов и битов масок достаточно при активном окне сопроцессора выбрать необходимый флаг, войти в меню окна и переключить выбранный флаг. Некоторые поля принимают более двух значений. Для них выполняется операция увеличения на единицу. Если перед выполнением операции поле имело максимальное значение, то оно обнулится.

Указатель команды содержит адрес команды, вызвавшей особый случай, а также код выполнявшейся операции. Если при возникновении особого случая использовался операнд, его адрес записывается в регистр указателя операнда.

Следующая группа команд позволяет оперировать со всеми регистрами управления сразу:

fstenv mem (float store environment) – записать содержимое всех нечисловых регистров сопроцессора в память;

fldenv mem (float load environment) – восстановить содержимое всех нечисловых регистров сопроцессора из памяти.

При работе в 16-разрядном реальном режиме область mem имеет формат, показанный на рис. 63.1.

Если возникает необходимость сохранить содержимое всех регистров, как числовых, так и нечисловых, то для этой цели можно использовать команду fsave (float save, сохранить среду сопроцессора). Для восстановления ранее запомненного состояния используется команда frstor (float restore, восстановить среду сопроцессора). В качестве операнда этих команд используется 94-байтовое поле памяти, имеющее следующее содержимое: регистр управления, регистр состояния, регистр признаков, указатель команды, указатель операнда, регистры общего назначения от ST до ST(7).

Раздел шестой

ЗАЩИЩЕННЫЙ РЕЖИМ

Статья 64. **Особенности 32-разрядных процессоров**

С появлением 32-разрядных процессоров корпорации Intel (80386, i486, Pentium) программисты значительно расширили спектр своих возможностей. Эти процессоры могут работать в трех режимах: реальном, защищенном и виртуального процессора 8086. Реальному режиму были посвящены первые 5 разделов этой книги. В этом и следующем разделе будет рассматриваться работа процессора в защищенном режиме.

Каждая следующая модель микропроцессора оказывается значительно совершеннее предыдущей. Так, начиная с процессора i486 арифметический сопроцессор, ранее выступавший в виде отдельной микросхемы, реализуется на одном кристалле с центральным процессором; улучшаются характеристики встроенной кеш-памяти; быстро растет скорость работы процессора. Однако все эти усовершенствования мало отражаются на принципах и методике программирования. Приводимые здесь программы будут одинаково хорошо работать на любом 32-разрядном процессоре. В дальнейшем под термином "процессор" мы будем понимать любую модификацию 32-разрядных процессоров корпорации Intel – от 80386 до различных вариантов процессоров Pentium, а также многочисленные разработки других фирм, совместимые с исходными процессорами Intel.

Процессор содержит около 40 программно-адресуемых регистров (не считая регистров сопроцессора), из которых 6 являются 16-разрядными, а большая часть остальных – 32-разрядными. Регистры принято объединять в группы: регистры данных, регистры-указатели, сегментные регистры, управляющие регистры, регистры системных адресов, отладочные регистры и регистры тестирования. Кроме того, в отдельную группу выделяют счетчик команд и регистр флагов. На рис. 64.1 приведены регистры, используемые в обычных (непривилегированных) прикладных программах.

Регистры общего назначения и регистры-указатели отличаются от аналогичных регистров МП 86 тем, что они являются 32-разрядными.

Для сохранения совместимости с ранними моделями процессоров допускается обращение к младшим половинам всех регистров, которые имеют те же мнемонические обозначения, что и в МП 86 (AX, BX, CX, DX, SI, DI, BP и SP). Естественно, сохранена возможность работы с младшими (AL, BL, CL и DL) и старшими (AH, BH, CH и DH) половинками регистров МП 86. Однако старшие половины 32-разрядных регистров процессора нс имеют мнемонических обозначений и непосредственно недоступны. Для того чтобы прочитать, например, содержимое старшей половины регистра EAX (биты 31...16), придется сдвинуть все содержимое EAX на 16 бит вправо (в регистр AX) и прочитать затем содержимое регистра AX.

Регистры данных

Биты	31 16	15 0	
EAX		AH AX AL	Аккумулятор
EBX		BH BX BL	Базовый регистр
ECX		CH CX CL	Счетчик
EDX		DH DX DL	Регистр данных

Регистры-указатели

Биты	31 16	15 0	
ESI		SI	Индекс источника
EDI		DI	Индекс приемника
EBP		BP	Указатель базы
ESP		SP	Указатель стека

Сегментные регистры

Биты	15 0	
CS		Регистр сегмента команд
DS		Регистр сегмента данных
ES		Регистр дополнительного сегмента данных
FS		Регистр дополнительного сегмента данных
GS		Регистр дополнительного сегмента данных
SS		Регистр сегмента стека

Указатель команд

Биты	31 16	15 0
EIP		IP

Рис. 64.1. Расширенные регистры 32-разрядных процессоров

Все регистры общего назначения и указатели программист может использовать по своему усмотрению для временного хранения адресов и данных размером от байта до двойного слова. Так, например, возможно использование следующих команд:

```
mov    EAX,0FFFFFFFFh;Работа с двойным словом (32 бита)
mov    BX,0FFFFh   ;Работа со словом (16 бит)
mov    CL,0FFh     ;Работа с байтом (8 бит)
```

Все сегментные регистры, как и в МП 86, являются 16-разрядными. В их состав включено еще два регистра – FS и GS, которые могут использоваться для хранения сегментных адресов двух дополнительных сегментов данных. Таким образом, при работе в реальном режиме из программы можно обеспечить доступ одновременно к четырем сегментам данных, а не к двум, как при использовании МП 86.

Регистр указателя команд также является 32-разрядным и обычно при описании процессора его называют EIP. Младшие 16 разрядов этого регистра соответствуют регистру IP МП 86.

Регистр флагов процессора 486 принято называть EFLAGS. Дополнительно к шести флагам состояния (CF, PF, AF, ZF, SF и OF) и трем флагам управления состоянием процессора (TF, IF и DF), назначение которых было описано в статье 3, он включает три новых флага NT, RF и VM и 2-байтовое поле IOPL (рис. 64.2).

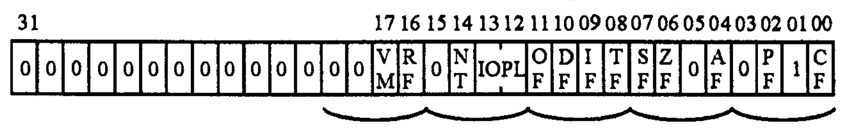


Рис. 64.2. Регистр флагов EFLAGS

Новые флаги NT, RF и VM и поле IOPL применяются процессором только в защищенном режиме.

Двухразрядное поле привилегий ввода-вывода IOPL (Input/Output Privilege Level) указывает на максимальное значение уровня текущего приоритета (от 0 до 3), при котором команды ввода-вывода выполняются без генерации исключительной ситуации.

Флаг вложенной задачи NT (Nested Task) показывает, является ли текущая задача вложенной в выполнение другой задачи. В этом случае NT=1. Флаг устанавливается автоматически при переключении задач. Значение NT проверяется командой iret для определения способа возврата в вызвавшую задачу.

Управляющий флаг рестарта RF (Restart Flag) используется совместно с отладочными регистрами. Если RF=1, то ошибки, возникшие во время отладки при исполнении команды, игнорируются до выполнения следующей команды.

Управляющий флаг виртуального режима VM (Virtual Mode) используется для перевода процессора из защищенного режима в режим виртуального МП 86. В этом случае процессор функционирует как быстродействующий МП 86, но реализует механизмы защиты памяти, страничной адресации и ряд других возможностей.

При работе с процессором программист имеет доступ к четырем управляющим регистрам CR0...CR3, в которых содержится информация о состоянии компьютера (рис. 64.3). Эти регистры доступны только в защищенном режиме для программ, имеющих уровень привилегий 0.

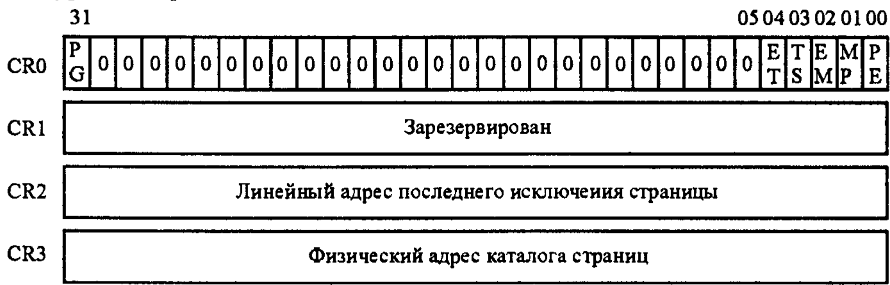


Рис. 64.3. Управляющие регистры процессора

Регистр CR0 представляет собой слово состояния системы. Для управления режимом работы процессора и указания его состояния используются следующие 6 бит регистра CR0:

- бит страничного преобразования PG (Paging Enable). Если этот бит установлен, то страничное преобразование разрешено; если сброшен, то запрещено;
- бит типа сопроцессора ET (Extension Type) в МП 80286 и 80386 указывал на тип подключенного сопроцессора. Если ET=1, то 80387, если ET=0, то 80287. В более новых процессорах бит ET всегда установлен;
- бит переключения задачи TS (Task Switched). Этот бит автоматически устанавливается процессором при каждом переключении задачи. Бит может быть очищен командой clts, которую можно использовать только на нулевом уровне привилегий;
- бит эмуляции сопроцессора EM (Emulate). Если EM=1, то обработка команд сопроцессора производится программно;
- бит присутствия арифметического сопроцессора MP (Math Present). Операционная система устанавливает MP=1, если сопроцессор присутствует. Этот бит управляет работой команды wait, используемой для синхронизации работы программы и сопроцессора;
- бит разрешения защиты PE (Protection Enable). При PE=1 процессор работает в защищенном режиме; при PE=0 – в реальном. PE может быть установлен при загрузке регистра CRO командами lmsw или mov CR0, а сброшен только командой mov CR0.

Регистр CR1 зарезервирован фирмой Intel для последующих моделей процессоров. Регистры CR2 и CR3 служат для поддержки страничного преобразования адреса. Эти два регистра используются вместе. CR2 содержит полный линейный адрес, вызвавший исключительную ситуацию на последней странице, а CR3 – адрес, указывающий базу каталога страницы.

Регистры системных адресов (рис. 64.4) используются в защищенном режиме работы процессора. Они задают расположение системных таблиц, служащих для организации сегментной адресации в защищенном режиме.

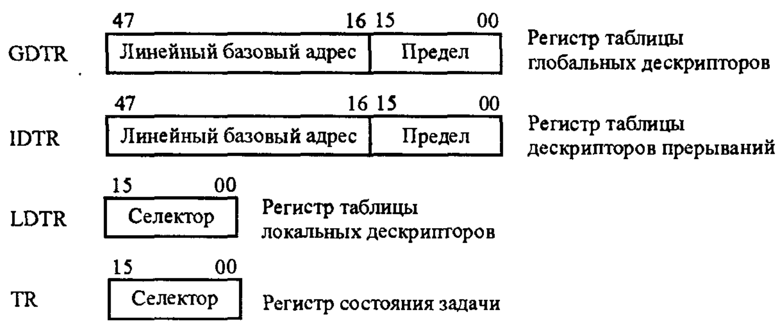


Рис. 64.4. Регистры системных адресов

В состав процессора входят 4 регистра системных адресов:

- GDTR (Global Descriptor Table Register) – регистр таблицы глобальных дескрипторов для хранения линейного базового адреса и границы таблицы глобальных дескрипторов;
- IDTR (Interrupt Descriptor Table Register) – регистр таблицы дескрипторов прерываний для хранения линейного базового адреса и границы таблицы дескрипторов прерываний;

- LDTR (Local Descriptor Table Register) – регистр таблицы локальных дескрипторов для хранения селектора сегмента таблицы локальных дескрипторов;
- TR (Task Register) – регистр состояния задачи для хранения селектора сегмента состояния задачи.

Отладочные регистры и регистры тестирования в настоящей книге не рассматриваются.

Статья 65. **Первое знакомство с защищенным режимом**

Как уже отмечалось, современные процессоры могут работать в трех режимах: реальном, защищенном и виртуального 86-го процессора. В реальном режиме процессоры функционируют фактически так же, как МП 86 с повышенным быстродействием и расширенным набором команд. Многие весьма привлекательные возможности процессоров принципиально не реализуются в реальном режимс, который сохраняется в современных процессорах лишь для обеспечения совместимости с предыдущими моделями процессоров. Все программы, приведенные в первых пяти разделах этой книги, относятся к реальному режиму и могут с равным успехом выполняться на любом из этих процессоров без каких-либо изменений. Характерной особенностью реального режима является ограничение объема адресуемой оперативной памяти величиной 1 Мбайт.

Только перевод микропроцессора в защищенный режим позволяет полностью реализовать все возможности, заложенные в его архитектуру и недоступные в реальном режиме. Сюда можно отнести:

- увеличение адресуемого пространства до 4 Гбайт;
- возможность работать в виртуальном адресном пространстве, превышающем максимально возможный объем физической памяти и достигающем огромной величины – 64 Тбайт. Правда, для реализации виртуального режима необходимы, помимо дисков большой емкости, еще и соответствующая операционная система, которая хранит все сегменты выполняемых программ в большом дисковом пространстве, автоматически загружая в оперативную память те или иные сегменты по мере необходимости;
- организация многозадачного режима с параллельным выполнением нескольких программ (процессов). Собственно говоря, многозадачный режим организует многозадачная операционная система, однако микропроцессор предоставляет необходимый для этого режима мощный и надежный механизм защиты задач друг от друга с помощью 4-уровневой системы привилсгий;
- страничная организация памяти, повышающая уровень защиты задач друг от друга и эффективность их выполнения.

При включении процессора в нем автоматически устанавливается режим реального адреса. Переход в защищенный режим осуществляется программно путем выполнения соответствующей последовательности команд. Поскольку многие детали функционирования процессора в реальном и защищенном режимах существенно раз-

личаются, программы, предназначенные для защищенного режима, должны быть написаны особым образом. Реальный и защищенный режимы несовместимы!

Архитектура современного микропроцессора необычайно сложна. Столь же сложными оказываются и программы, использующие средства защищенного режима. К счастью, однако, отдельные архитектурные особенности защищенного режима оказываются в достаточной степени замкнутыми и не зависящими друг от друга. Так, при работе в однозадачном режиме отпадает необходимость в изучении многообразных и замысловатых методов взаимодействия задач; во многих случаях можно отключить (или, точнее, не включать) механизм страничной организации памяти; часто нет необходимости использовать уровни привилегий. Все эти ограничения существенно упрощают освоение защищенного режима.

Начнем изучение защищенного режима с рассмотрения простейшей (но, к сожалению, все же весьма сложной) программы, которая, будучи запущена обычным образом под управлением MS-DOS, переключает процессор в защищенный режим, выводит на экран для контроля несколько символов, переходит назад в реальный режим и завершается стандартным для DOS образом (пример 65.1). Рассматривая эту программу, мы познакомимся с основополагающей особенностью защищенного режима – сегментной адресацией памяти, которая осуществляется совсем не так, как в реальном режиме.

Следует заметить, что для выполнения программ этого раздела книги необходимо, чтобы на компьютере была установлена система MS-DOS "в чистом виде" (не в виде сеанса DOS системы Windows). Кроме этого, перед запуском программ защищенного режима следует выгрузить драйверы обслуживания расширенной памяти HIMEM.SYS и EMM386.EXE. Для подготовки программ к выполнению нами использовались транслятор TASM и компоновщик TLINK из пакета TASM, хотя с таким же успехом можно воспользоваться любым другим ассемблером, например пакетом Microsoft MASM.

Пример 65.1. Переход в защищенный режим и обратно

```
.586P          (1);Разрешение трансляции всех команд Pentium
;Структура для описания дескрипторов сегментов
descr   struc               ;(2)Начало объявления структуры
lim     dw 0                ;(3)Граница (биты 0…15)
base_l  dw 0                ;(4)База, биты 0…15
base_m  db 0                ;(5)База, биты 1…23
attr_1  db 0                ;(6)Байт атрибутов 1
attr_2  db 0                ;(7)Граница (биты 16…19) и атрибуты 2
base_h  db 0                ;(8)База, биты 24…31
descr   ends                ;(9)Конец объявления структуры
;Сегмент данных
data     segment use16      ;(10)16-разрядный сегмент
;Таблица глобальных дескрипторов GDT
gdt_null descr <0,0,0,0,0,0> ;(11)Селектор 0, нулевой дескриптор
gdt_data descr <data_size-1,0,0,92h,0,0> ;(12)Селектор 8, сегмент данных
gdt_code descr <code_size-1,0,0,98h,0,0> ;(13)Селектор 16, сегмент команд
gdt_stack descr <255,0,0,92h,0,0> ;(14)Селектор 24, сегмент стека
gdt_screen descr <3999,8000h,0Bh,92h,0,0> ;(15)Селектор 32, видеопамять
gdt_size=$-gdt_null         ;(16)Размер GDT
;Различные данные программы
pdescr  df 0                ;(17)Псевдодескриптор для команды lgdt
sym     db 1                ;(18)Символ для вывода на экран
attr    db 1Eh              ;(19)Его атрибут
msg db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$' ;(20)
data_size=$-gdt_null        ;(21)Размер сегмента данных
```

```
data     ends                  ;(22)
;Сегмент команд
text     segment use16       ;(23)16-разрядный сегмент
         assume CS:text,DS:data ;(24)
main     proc                  ;(25)
         xor    EAX,EAX        ;(26)Очистим EAX
         mov    AX,data        ;(27)Загрузим в DS сегментный
         mov    DS,AX          ;(28)адрес сегмента данных
;Вычислим 32-битовый линейный адрес сегмента данных и загрузим его
;в дескриптор сегмента данных в таблице глобальных дескрипторов GDT
         shl    EAX,4          ;(29)EAX=линейный базовый адрес
         mov    EBP,EAX        ;(30)Сохраним его в EBP для будущего
         mov    BX,offset gdt_data;(31)BX=смещение дескриптора
         mov    [BX].base_l,AX;(32)Загрузим младшую часть базы
         shr    EAX,16         ;(33)Старшую половину EAX в AX
         mov    [BX].base_m,AL;(34)Загрузим среднюю часть базы
;Вычислим и загрузим в GDT линейный адрес сегмента команд
         xor    EAX,EAX        ;(35)Очистим EAX
         mov    AX,CS          ;(36)Сегментный адрес сегмента команд
         shl    EAX,4          ;(37)
         mov    BX,offset gdt_code;(38)
         mov    [BX].base_l,AX;(39)
         shr    EAX,16         ;(40)
         mov    [BX].base_m,AL;(41)
;Вычислим и загрузим в GDT линейный адрес сегмента стека
         xor    EAX,EAX        ;(42)Очистим EAX
         mov    AX,SS          ;(43)Сегментный адрес сегмента стека
         shl    EAX,4          ;(44)
         mov    BX,offset gdt_stack;(45)
         mov    [BX].base_l,AX;(46)
         shr    EAX,16         ;(47)
         mov    [BX].base_m,AL;(48)
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
         mov    dword ptr pdescr+2,EBP;(49)База GDT
         mov    word ptr pdescr,gdt_size-1;(50)Граница GDT
         lgdt   pdescr         ;(51)Загрузим регистр GDTR
;Подготовимся к возврату из защищенного режима в реальный
         mov    AX,40h         ;(52)Настроим ES на область
         mov    ES,AX          ;(53)данных BIOS
         mov    word ptr ES:[67h],offset return;(54)Смещение возврата
         mov    ES:[69h],CS ;(55)Сегмент возврата
         mov    AL,0Fh         ;(56)Выборка байта состояния отключения
         out    70h,AL         ;(57)Порт КМОП-микросхемы
         mov    AL,0Ah         ;(58)Установка режима восстановления
         out    71h,AL         ;(59)в регистре 0Fh сброса процессора
         cli                   ;(60)Запрет аппаратных прерываний
;Переходим в защищенный режим
         mov    EAX,CR0        ;(61)Получим содержимое регистра CR0
         or     EAX,1          ;(62)Установим бит защищенного режима
         mov    CR0,EAX        ;(63)Запишем назад в CR0
;---------------------------------------------;
; Теперь процессор работает в защищенном режиме ;
;---------------------------------------------;
;Загружаем в CS:IP селектор:смещение точки continue
         db 0EAh               ;(64)Код команды far jmp
         dw offset continue;(65)Смещение
         dw 16                 ;(66)Селектор сегмента команд
continue:                      ;(67)
;Делаем адресуемыми данные
         mov    AX,8           ;(68)Селектор сегмента данных
         mov    DS,AX          ;(69)
```

```
;Делаем адресуемым стек
        mov   AX,24          ;(70)Селектор сегмента стека
        mov   SS,AX          ;(71)
;Инициализируем ES
        mov   AX,32          ;(72)Селектор сегмента видеобуфера
        mov   ES,AX          ;(73)Инициализируем ES
;Выводим на экран тестовую строку символов
        mov   DI,1920        ;(74)Начальная позиция на экране
        mov   CX,80          ;(75)Число выводимых символов
        mov   AX,word ptr sym;(76)Символ+атрибут
scrn:   stosw                ;(77)Содержимое AX на экран
        inc   AL             ;(78)Инкремент кода символа
        loop  scrn           ;(79)Цикл вывода
;Вернемся в реальный режим
        mov   AL,0FEh        ;(80)Команда сброса процессора
        out   64h,AL         ;(81)в порт 64h
        hlt                  ;(82)Останов процессора до окончания сброса
;-----------------------------------------------;
; Теперь процессор снова работает в реальном режиме ;
;-----------------------------------------------;
return:                      ;(83)
;Восстановим вычислительную среду реального режима
        mov   AX,data        ;(84)Сделаем адресуемыми данные
        mov   DS,AX          ;(85)
        mov   AX,stk         ;(86)Сделаем адресуемым стек
        mov   SS,AX          ;(87)
        mov   SP,256         ;(88)Настроим SP
        sti                  ;(89)Разрешим аппаратные прерывания
;Работаем в DOS
        mov   AH,09h         ;(90)Проверим выполнение функций DOS
        mov   DX,offset msg;(91)после возврата в реальный режим
        int   21h            ;(92)
        mov   AX,4C00h       ;(93)Завершим программу обычным образом
        int   21h            ;(94)
main    endp                 ;(95)
code_size=$-main             ;(96)Размер сегмента команд
text    ends                 ;(97)
;Сегмент стека
stk     segment stack use16;(98)16-разрядный сегмент
        db 256 dup ('^')     ;(99)
stk     ends                 ;(100)
        end main             ;(101)
```

Тридцатидвухразрядные микропроцессоры отличаются расширенным набором команд, часть которых относится к привилегированным. Для того чтобы разрешить транслятору обрабатывать эти команды, в текст программы включена директива ассемблера .586P. Поскольку используемые в этом и последующих примерах привилегированные команды относятся к базовым командам защищенного режима, характерным не только для процессоров Pentium, но и для более ранних 32-разрядных процессоров, с таким же успехом можно использовать директивы .486P или .386P.

Программа начинается с описания структуры дескриптора сегмента. В отличие от реального режима, в котором сегменты определяются их базовыми адресами, задаваемыми программистом в явной форме, в защищенном режиме для каждого сегмента программы должен быть определен дескриптор – 8-байтовое поле, в котором в определенном формате записываются базовый адрес сегмента, его длина и некоторые другие характеристики (рис. 65.1).

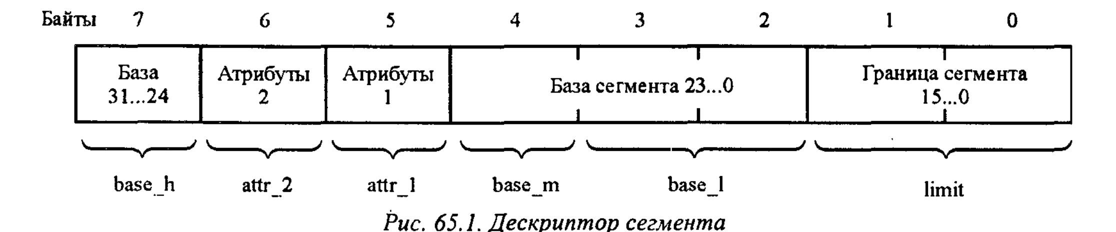


Рис. 65.1. Дескриптор сегмента

Теперь для обращения к требуемому сегменту программист заносит в сегментный регистр не сегментный адрес, а так называемый селектор (рис. 65.2), в состав которого входит номер (индекс) соответствующего сегменту дескриптора. Процессор по этому номеру находит нужный дескриптор, извлекает из него базовый адрес сегмента и, прибавляя к нему указанное в конкретной команде смещение (относительный адрес), формирует адрес ячейки памяти. Индекс дескриптора (0, 1, 2 и т. д.) записывается в селектор начиная с бита 3, что эквивалентно умножению его на 8. Таким образом, можно считать, что селекторы последовательных дескрипторов представляют собой числа 0, 8, 16, 24 и т. д. (см. комментарии к предложениям 11...15). Другие поля селектора, которые для нашего случая принимают значение 0, будут описаны позже.

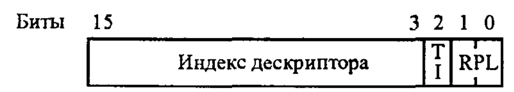


Рис. 65.2. Селектор дескриптора

Структура descr предоставляет шаблон для дескрипторов сегментов, облегчающий их формирование. Вообще структура представляет собой формальное описание формата произвольного набора данных, который можно накладывать на различные участки памяти, чтобы затем обращаться к полям этих участков с помощью мнемонических имен, определенных в описании структуры. Следует подчеркнуть, что описание в программе структуры не приводит к выделению для нее памяти; транслятор (ассемблер) просто запоминает имя и состав структуры для дальнейшего использования. Соответствующий структуре объем памяти выделяется (в сегменте данных) лишь в том случае, если имя структуры вместе с набором фактических параметров указывается программистом в полях сегмента, как это сделано в предложениях 11...15 нашего примера. Для обращения к этим наборам данных им разумно назначить имена (у нас это gdt_null, gdt_data и т. д.). Сравнивая описание структуры descr в программе с рис. 65.1, нетрудно проследить их соответствие друг другу.

Рассмотрим (сначала вкратце) содержимое дескриптора. Граница (limit) сегмента представляет собой номер последнего байта сегмента. Так, для сегмента размером 375 байт граница равна 374. Поле границы состоит из 20 бит и разбито на две части. Как видно из рис. 65.1, младшие 16 бит границы занимают байты 0 и 1 дескриптора, а старшие 4 бита входят в байт атрибутов 2, занимая в нем биты 0...3 (рис. 65.3). Получается, что размер сегмента ограничен величиной 1 Мбайт. На самом деле это не так. Граница может указываться либо в байтах (и тогда, действительно, максимальный размер сегмента равен 1 Мбайт), либо в блоках по 4 Кбайт (и тогда размер сегмента может достигать 4 Гбайт). В каких единицах задается граница – определяет старший бит байта атрибутов 2, называемый битом дробности. Если он равен нулю, граница указывается в байтах; если единице – в блоках по 4 Кбайт. Назначение других полей байта атрибутов 2 будет пояснено позже.

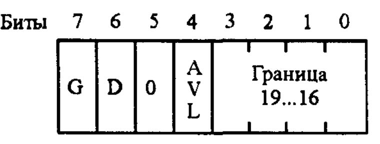


Рис. 65.3. Байт атрибутов 2

База сегмента (32 бита) определяет начальный линейный адрес сегмента в адресном пространстве процессора. Линейным называется адрес, выраженный не в виде комбинации *сегмент:смещение*, а просто номером байта в адресном пространстве. Казалось бы, линейный адрес – это просто другое название физического адреса. Для нашего примера это так и есть, в нем линейные адреса совпадают с физическими. Если, однако, в процессоре включено страничное преобразование памяти, то процедура преобразования адресов усложняется. Отдельные блоки размером 4 Кбайт (страницы) линейного адресного пространства могут произвольным образом отображаться на физические адреса, в частности и так, что большие линейные адреса отображаются на начало физической памяти и наоборот. Страничная адресация осуществляется аппаратно (хотя для ее включения требуются определенные программные усилия) и действует независимо от сегментной организации программы. Поэтому во всех программных структурах защищенного режима фигурируют не физические, а линейные адреса. Если страничная адресация выключена, эти линейные адреса совпадают с физическими, если включена – могут и не совпадать. Более подробно о страничном преобразовании будет рассказано в статье 73.

Поскольку в дескриптор записывается 32-битовый линейный базовый адрес (номер байта), сегмент в защищенном режиме может начинаться на любом байте, а не только на границе параграфа, и располагаться в любом месте адресного пространства 4 Гбайт.

Поле базы, как и поле границы, разбито на две части: биты 0...23 занимают байты 2…4 дескриптора, а биты 24...31 – байт 7. Для удобства программного обращения в структуре descr база описывается тремя полями: младшим словом (base_l) и 2 байтами: средним (base_m) и старшим (base_h).

В байте атрибутов 1 задается ряд характеристик сегмента. Не вдаваясь пока в подробности этих характеристик, укажем, что в примере 65.1 используются сегменты двух типов: сегмент команд, для которого байт attr_1 должен иметь значение 98h, и сегмент данных (или стека) с кодом 92h.

Некоторые дополнительные характеристики сегмента указываются в старшем полубайте байта attr_2 (в частности, тип дробности). Для всех наших сегментов значение этого полубайта равно нулю.

Сегмент данных data, который для удобства изучения программы расположен в ее начале, до сегмента команд, объявлен с типом использования use16 (так же будут объявлены и оба остальных сегмента программы). Этот описатель объявляет, что в данном сегменте будут по умолчанию использоваться 16-битовые адреса. Если бы мы готовили нашу программу для работы под управлением операционной системы защищенного режима, реализующей все возможности микропроцессора, тип использования был бы use32. Однако наша программа будет запускаться под управлением DOS, которая работает в реальном режиме с 16-битовыми адресами и операндами.

Сегмент данных начинается с описания важнейшей системной структуры – таблицы глобальных дескрипторов. Как уже отмечалось выше, обращение к сегментам в защищенном режиме возможно исключительно через дескрипторы этих сегментов. Та-

ким образом, в таблице дескрипторов должно быть описано столько дескрипторов, сколько сегментов использует программа. В нашем случае в таблицу включены, помимо обязательного нулевого дескриптора, всегда занимающего первое место в таблице, 4 дескриптора для сегментов данных, команд, стека и дополнительного сегмента данных, который мы наложим на видеопамять, чтобы обеспечить возможность вывода в нее символов. Порядок дескрипторов в таблице (кроме нулевого) не имеет значения.

Помимо единственной таблицы глобальных дескрипторов, обозначаемой GDT (Global Descriptor Table), в памяти может находиться множество таблиц локальных дескрипторов – LDT (Local Descriptor Table). Разница между ними в том, что сегменты, описываемые глобальными дескрипторами, доступны всем задачам, выполняемым процессором, а к сегментам, описываемым локальными дескрипторами, может обращаться только та задача, в которой эти дескрипторы описаны. Поскольку пока мы имеем дело с однозадачным режимом, локальная таблица нам не нужна.

Поля дескрипторов для наглядности заполнены конкретными данными явным образом, хотя объявление структуры descr с нулями во всех полях позволяет описать дескрипторы несколько короче, например:

```
gdt_null descr<>           ;Селектор 0, нулевой дескриптор
gdt_data descr<data_size-1,,,92h>;Селектор 8, сегмент данных
```

В дескрипторе gdt_data, описывающем сегмент данных программы, заполняется поле границы сегмента (фактическое значение размера сегмента data_size вычисляется транслятором, см. предложение 21), а также байт атрибутов 1. Код 92h говорит о том, что это сегмент данных с разрешением записи и чтения. Базу сегмента, т. е. физический адрес его начала, придется вычислить программно и занести в дескриптор уже на этапе выполнения.

Дескриптор gdt_code сегмента команд заполняется схожим образом. Код атрибута 98h обозначает, что это исполняемый сегмент, к которому, между прочим, запрещено обращение с целью чтения или записи. Таким образом, сегменты команд в защищенном режиме нельзя модифицировать по ходу выполнения программы.

Дескриптор gdt_stack сегмента стека имеет, как и любой сегмент данных, код атрибута 92h, что разрешает его чтение и запись, и явным образом заданную границу 255 байт, что соответствует размеру стека. Базовый адрес сегмента стека также будет вычислен на этапе выполнения программы.

Последний дескриптор, gdt_screen описывает страницу 0 видеопамяти. Размер видеостраницы, как известно, составляет 4000 байт, поэтому в поле границы указано число 3999. Базовый физический адрес страницы тоже известен, он равен B8000h. Младшие 16 бит базы (число 8000h) заполняют слово base_l дескриптора, биты 16...19 (число 0Bh) – байт base_m. Биты 20...31 базового адреса равны нулю, поскольку видеопамять размещается в первом мегабайте адресного пространства.

Перед переходом в защищенный режим процессору надо будет сообщить физический адрес таблицы глобальных дескрипторов и ее размер (точнее, границу). Размер GDT определяется на этапе трансляции в предложении 16.

Назначение оставшихся строк сегмента данных станет ясным в процессе рассмотрения программы.

Сегмент команд text начинается, как и всегда, оператором segment, в котором указывается тип использования use16, так как мы составляем 16-разрядное приложение. Указание описателя use16 не запрещает использовать в программе 32-битовые регистры.

В предложении 26 очищается регистр EAX, после чего с помощью регистра AX обычным образом инициализируется сегментный регистр DS для работы в реальном режиме.

Фактически весь остальной текст программы примера 65.1, кроме ее завершающих строк, а также фрагмента, выполняемого в защищенном режиме, посвящен подготовке перехода в защищенный режим. Прежде всего надо завершить формирование дескрипторов сегментов программы, в которых остались незаполненными базовые адреса сегментов. Линейные (32-битовые) адреса определяются путем умножения значений сегментных адресов на 16.

Содержимое EAX, в котором сейчас находится сегментный адрес сегмента данных, командой shl сдвигается влево на 4 бита, образуя линейный 32-битовый адрес. Поскольку этот адрес будет использоваться в последующих фрагментах программы, он сохраняется в регистре EBP (или в любом другом свободном регистре общего назначения). В BX загружается смещение дескриптора сегмента данных, после чего в дескриптор из регистра AX заносится младшая половина линейного адреса. Поскольку к старшей половине регистра EAX (где нас интересуют разряды 16...23) обратиться невозможно, все содержимое EAX с помощью команды shr сдвигается вправо на 16 бит, в результате чего старшая половина линейного адреса попадает в регистр AX.

После сдвига содержимое AL (где теперь находятся биты 16...23 линейного адреса) заносится в поле base_m дескриптора. Вообще говоря, биты 24...31 линейного адреса следовало отправить в поле base_h, однако все поля нашей программы, включая видеопамять, расположены в первом мегабайте и старшие 8 бит линейных адресов равны нулю.

Аналогично вычисляются и заносятся в соответствующие поля дескрипторов gdt_code и gdt_stack 32-битовые линейные адреса сегментов команд и стека.

Следующий этап подготовки к переходу в защищенный режим – загрузка в регистр процессора GDTR (Global Descriptor Table Register, регистр таблицы глобальных дескрипторов) информации о таблице глобальных дескрипторов. Эта информация включает в себя линейный базовый адрес таблицы и ее границу и размещается в 6 байтах поля данных, называемого псевдодескриптором. Для загрузки GDTR предусмотрена специальная привилегированная команда lgdt (load global descriptor table, загрузка таблицы глобальных дескрипторов), которая требует указания в качестве операнда имени псевдодескриптора. Формат псевдодескриптора приведен на рис. 65.4; 6-байтовое поле pdescr для псевдодескриптора выделяется в сегменте данных с помощью оператора ассемблера df.

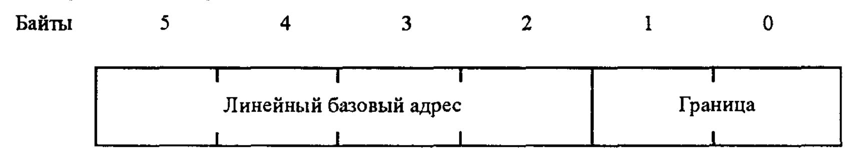


Рис. 65.4. Формат псевдодескриптора

В нашем примере заполнение псевдодескриптора упрощается вследствие того, что таблица глобальных дескрипторов расположена в начале сегмента данных и ее базовый адрес совпадает с базовым адресом всего сегмента, который уже был вычислен и помещен в дескриптор gdt_data. В предложении 49 в псевдодескриптор заносится линейный адрес GDT, а в предложении 50 заполняется поле границы. Команда

```
lgdt    pdescr
```

загружает регистр GDTR и сообщает процессору о местонахождении и размере GDT.

В принципе теперь можно перейти в защищенный режим. Однако мы запускаем нашу программу под управлением DOS (а как еще мы можем ее запустить?); и естественно завершить ее также обычным образом, чтобы не нарушить работоспособность системы. Но в защищенном режиме запрещены любые обращения к функциям DOS или BIOS. Причина этого совершенно очевидна – и DOS и BIOS являются программами реального режима, в которых широко используется сегментная адресация реального режима, т. е. загрузка в сегментные регистры сегментных адресов. В защищенном же режиме в сегментные регистры загружаются не сегментные адреса, а селекторы. Кроме того, обращение к функциям DOS и BIOS осуществляется с помощью команд int с определенными номерами, а в защищенном режиме эти команды приведут к совершенно иным результатам. Таким образом, программу, работающую в защищенном режиме, нельзя завершить средствами DOS. Сначала ее надо вернуть в реальный режим.

Возврат в реальный режим можно осуществить сбросом процессора. Действия процессора после сброса определяются одной из ячеек КМОП-микросхемы – байтом состояния отключения, располагаемым по адресу Fh. Возможные значения байта состояния отключения приведены в табл. 65.1. В частности, если в этом байте записан код 0Ah, после сброса управление немедленно передается по адресу, который извлекается из двухсловной ячейки 40h:67h, расположенной в области данных BIOS. Таким образом, для подготовки возврата в реальный режим мы должны в ячейку 40h:67h записать адрес возврата, а в байт Fh КМОП-микросхемы занести код Ah. В предложениях 52...55 полный адрес точки возврата return заносится по адресу 40h:67h. Точка возврата может располагаться в любом месте программы.

Таблица 65.1. Коды байта сброса (перехода в реальный режим)

Коды	*Назначение*
0	По Ctrl+Alt+Del осуществляется перезагрузка без POST
1	Сброс после определения размера памяти
2	Сброс после теста памяти
3	Сброс по ошибке памяти
4	Сброс с Int 19h (перезагрузка без очистки памяти и восстановления векторов)
5	Возврат в реальный режим с помощью jmp far [40:67] по команде сброса. Контроллеры прерываний перепрограммируются
6, 7, 8	Сброс после выполнения тестов защищенного режима
9	Сброс после выполнения Int 15h, AH=87h (пересылка в расширенную память из основной)
0Ah	Возврат в реальный режим с помощью Jmp far [40:67] по команде сброса. Контроллеры прерываний не перепрограммируются

В предложениях 56...59 в порт 70h засылается код 0Fh, который выбирает для записи байт Fh КМОП-микросхемы, а затем в порт данных 71h посылается код 0Ah, определяющий режим восстановления (переход по адресу, извлекаемому из области данных BIOS).

Еще одна важная операция, которую необходимо выполнить перед переходом в защищенный режим, заключается в запрете всех аппаратных прерываний. Дело в том, что в защищенном режиме процессор выполняет процедуру прерывания не так, как в реальном. При поступлении сигнала прерывания процессор не обращается к таблице

векторов прерываний в первом килобайте памяти, как в реальном режиме, а извлекает адрес программы обработки прерывания из таблицы дескрипторов прерываний, построенной схоже с таблицей глобальных дескрипторов и располагаемой в программе пользователя (или в операционной системе). В примере 65.1 такой таблицы нет, и на время работы нашей программы прерывания придется запретить. Запрет всех аппаратных прерываний осуществляется командой cli (предложение 60).

В предложениях 61...63 осуществляется перевод процессора в защищенный режим, для чего достаточно установить бит PM в управляющем регистре процессора CR0 (см. рис. 64.3). При включении процессора этот бит сбрасывается и в процессоре устанавливается реальный режим. Установка в 1 младшего бита CR0 переводит процессор в защищенный режим, сброс этого бита возвращает процессор в режим реальных адресов.

Обращение к регистрам управления осуществляется исключительно с помощью команды mov, причем в качестве второго операнда должен быть указан регистр общего назначения. Для модификации CR0 его содержимое сначала считывается в EAX, там с помощью команды or устанавливается младший бит, после чего второй командой mov новое значение загружается в CR0. Процессор переходит в защищенный режим.

Хотя защищенный режим установлен, однако действия по настройке системы еще не закончены. Действительно, во всех используемых в программе сегментных регистрах хранятся не селекторы дескрипторов сегментов, а базовые сегментные адреса, не имеющие смысла в защищенном режиме. Между прочим, отсюда можно сделать вывод, что после перехода в защищенный режим программа не должна работать, так как в регистре CS пока еще нет селектора сегмента команд и процессор не может обращаться к этому сегменту. В действительности это не совсем так.

В процессоре для каждого из сегментных регистров имеется так называемый теневой регистр дескриптора, который имеет формат дескриптора (рис. 65.5). Теневые регистры недоступны программисту; они автоматически загружаются процессором из таблицы дескрипторов каждый раз, когда процессор инициализирует соответствующий сегментный регистр. Таким образом, в защищенном режиме программист имеет дело с селекторами, т. е. номерами дескрипторов, а процессор – с самими дескрипторами, хранящимися в теневых регистрах. Именно содержимое теневого регистра (в первую очередь линейный адрес сегмента) определяет область памяти, к которой обращается процессор при выполнении конкретной команды.

В реальном режиме теневые регистры заполняются не из таблицы дескрипторов, а непосредственно самим процессором. В частности, процессор заполняет поле базы каждого теневого регистра линейным базовым адресом сегмента, полученным путем умножения на 16 содержимого сегментного регистра, как это и положено в реальном режиме. Поэтому после перехода в защищенный режим в теневых регистрах находятся правильные линейные базовые адреса и программа будет выполняться правильно, хотя с точки зрения правил адресации защищенного режима содержимое сегментных регистров лишено смысла.

Тем не менее после перехода в защищенный режим прежде всего следует загрузить в используемые сегментные регистры (и в частности, в регистр CS) селекторы соответствующих сегментов. Это позволит процессору правильно заполнить все поля теневых регистров из таблицы дескрипторов. Пока эта операция не выполнена, некоторые поля теневых регистров (в частности, границы сегментов) могут содержать неверную информацию.

Сегментные регистры		Теневые регистры		
CS	Селектор	База	Граница	Атрибуты
DS	Селектор	База	Граница	Атрибуты
ES	Селектор	База	Граница	Атрибуты
FS	Селектор	База	Граница	Атрибуты
GS	Селектор	База	Граница	Атрибуты
SS	Селектор	База	Граница	Атрибуты

Рис. 65.5. Сегментные и теневые регистры

Загрузить селекторы в сегментные регистры DS, SS и ES не представляет труда (предложения 68...73). Но как загрузить селектор в регистр CS, к которому недопустимо прямое программное обращение? Для этого можно воспользоваться искусственно сконструированной командой дальнего перехода, которая, как известно, приводит к смене содержимого и IP и CS. Предложения 64...66 демонстрируют эту методику. В реальном режиме мы поместили бы во второе слово адреса сегментный адрес сегмента команд, в защищенном же мы записываем в него селектор этого сегмента (число 16).

Следующий фрагмент примера 65.1 (предложения 72...79) является чисто иллюстративным. В нем после инициализации сегментного регистра ES в видеопамять выводится последовательность цветных символов, чем подтверждается правильное функционирование программы в защищенном режиме.

Код первого символа выводимой последовательности и его атрибут заносятся в регистр AX из ячеек sym и attr сегмента данных. Содержимое AL в цикле вывода инкрементируется, что приводит к отображению на экране последовательности первых 80 символов кодовой таблицы. Прибегать здесь к услугам сегмента данных не было необходимости; проще было занести коды символа и атрибута в регистр AX непосредственно:

```
mov   AX,1E01h
```

после чего выполнять инкремент регистра AL. Мы, однако, поместили выводимые данные в сегмент данных умышленно, чтобы обратиться к этому сегменту в защищенном режиме и удостовериться в том, что это обращение выполняется правильно.

Как уже отмечалось выше, для того чтобы не нарушить работоспособность DOS, процессор следует вернуть в реальный режим, после чего можно будет завершить программу обычным образом. Перейти в реальный режим можно разными способами; в нашем примере сброс процессора выполняется засылкой команды FEh в порт 64h контроллера клавиатуры (предложения 80 и 81). Эта команда возбуждает сигнал на одном из выводов контроллера клавиатуры, который в конечном счете приводит к появлению сигнала сброса на выводе RESET микропроцессора.

После сброса процессор работает в реальном режиме, причем управление передается одной из программ BIOS, которая анализирует содержимое байта состояния отключения (0Fh) КМОП-микросхемы и, поскольку мы записали туда код 0Ah, осущест-

вляет передачу управления по адресу, хранящемуся в ячейке 40h:67h области данных BIOS. В нашем случае переход осуществляется на метку return.

Команда hlt (halt, останов) позволяет организовать ожидание сброса процессора, который выполняется не мгновенно. Вместо команды hlt можно было использовать бесконечный цикл:

```
stop:   jmp   stop
```

Если команду ожидания опустить, процессор до своего останова успеет выполнить несколько следующих команд, после чего перейдет на метку return.

Передача управления на метку return осуществляется программами BIOS, которые, естественно, используют регистры процессора. В частности, регистры SS:SP и DS указывают на поля данных BIOS, и их следует инициализировать заново, что и выполняется в предложениях 84...88.

Для восстановления работоспособности системы следует разрешить прерывания (предложение 89), после чего программа может продолжаться уже в реальном режиме. В рассматриваемом примере для проверки работоспособности системы на экран выводится некоторый текст с помощью функции DOS 09h. Для наглядности в него включены Esc-последовательности смены цвета символов, поэтому программу следует выполнять при установленном драйвере ANSI.SYS.

Программа завершается обычным образом функцией DOS 4Ch. Нормальное завершение программы и переход в DOS тоже в какой-то мере свидетельствует о ее правильности.

У рассмотренной программы имеется серьезный недостаток – полное отсутствие средств отладки. Для отладки программ защищенного режима используется механизм исключений, в нашей же программе этот механизм не активизирован. Поэтому всякие неполадки при работе в защищенном режиме, которые с помощью указанного механизма можно было бы обнаружить и проанализировать, в данном случае будут приводить к сбросу процессора. Однако после сброса программа скорее всего будет работать правильно: на экран будет выведена запланированная строка и программа завершится с передачей управления DOS. Таким образом, критерием программных ошибок защищенного режима может служить правильная в целом работа программы при отсутствии на экране цветных символов, которые должны выводиться в защищенном режиме. В следующей статье мы дополним программу простыми средствами диагностики, которые в дальнейшем помогут в исследовании защищенного режима и отладке примеров.

Остановимся еще на вопросе о способах перехода из защищенного режима в реальный. В рассмотренном примере перевод процессора в защищенный режим осуществлялся путем установки бита PE (бит 0) в регистре CR0, однако для возврата в реальный режим использовался более грубый способ – сброс процессора. Почему нельзя было вернуться в реальный режим попросту сбросив бит PE в CR0? В действительности так сделать можно, однако такое переключение режима потребует выполнения целого ряда дополнительных операций. Рассмотрение этих операций позволит нам глубже вникнуть в различия реального и защищенного режимов.

При работе в защищенном режиме в дескрипторах сегментов записаны, среди прочего, их линейные адреса и границы. Процессор при выполнении команды с адресацией к тому или иному сегменту сравнивает полученный им относительный адрес с границей сегмента и, если команда пытается адресоваться за пределами сегмента, формирует прерывание (исключение) нарушения общей защиты. Если в программе пре-

дусмотрена обработка исключений, такую ситуацию можно обнаружить и как-то исправить. Таким образом, в защищенном режиме программа не может выйти за пределы объявленных ею сегментов, а также не может выполнить действия, запрещенные атрибутами сегмента. Так, если сегмент объявлен исполняемым (код атрибута 1 98h), то данные из этого сегмента нельзя читать или модифицировать; если атрибут сегмента равен 92h, то в таком сегменте не может быть исполняемых команд, на зато данные можно как читать, так и модифицировать. Указав для какого-то сегмента код атрибута 90h, мы получим сегмент с запрещением записи. При попытке записи в этот сегмент процессор сформирует исключение общей защиты.

Как уже отмечалось, дескрипторы сегментов хранятся в процессе выполнения программы в теневых регистрах, которые загружаются автоматически при записи в сегментный регистр селектора.

При работе в реальном режиме некоторые поля теневых регистров должны быть заполнены вполне определенным образом. Так, для сегментов данных и стека (адресуемых через сегментные регистры DS, ES и SS) должны быть установлены следующие характеристики: граница=FFFFh, бит дробности=0, доступ для записи разрешен. Сегмент команд должен быть объявлен исполняемым с той же границей FFFFh.

Приведенный перечень ограничений не полон; мы рассматриваем здесь только описанные ранее атрибуты сегментов. Следует также заметить, что в 32-разрядных процессорах имеются 6 сегментных регистров (соответственно, и 6 теневых регистров для хранения их дескрипторов, см. рис. 65.5). Отмеченные выше ограничения относятся к сегментам, адресуемым через любой из этих регистров. Наконец, стоит подчеркнуть, что границы всех сегментов должны быть точно равны FFFFh; любое другое число, например FFFEh, "не устроит" реальный режим.

Если мы просто перейдем в реальный режим сбросом бита 0 в регистре CR0, то в теневых регистрах останутся дескрипторы защищенного режима и при первом же обращении к любому сегменту программы возникнет исключение общей защиты, так как ни один из наших сегментов не имеет границы, равной FFFFh. Поскольку мы не обрабатываем исключения, произойдет сброс процессора и перезагрузка компьютера, если мы заранее не настроим байт состояния отключения и ячейки области данных BIOS, как это было сделано в примере 65.1. Таким образом, перед переходом в реальный режим необходимо исправить дескрипторы всех наших сегментов: команд, данных, стека и видеопамяти. К сегментным регистрам FS и GS мы не обращались, и о них можно не заботиться.

Рассмотрим модификацию примера 65.1 (назовем ее примером 65.2), в которой выполнен более изящный, хотя и более трудоемкий переход в реальный режим путем сброса бита PE в регистре CR0. Изменению подвергнутся лишь два фрагмента программы.

Прежде всего из программы можно исключить весь блок подготовки к возврату из защищенного режима в реальный (предложения 52...59 примера 65.1). Следующие далее блоки перехода в защищенный режим и работы в нем останутся без изменений.

Блок возврата в защищенный режим, который в примере 65.1 занимал всего три строки (предложения 80...82), теперь примет вид, представленный в примере 65.2.

Пример 65.2. Модификация примера 65.1 (фрагмент возврата в защищенный режим)

```
;Вернемся в реальный режим
;Сформируем и загрузим дескрипторы для реального режима
        mov     gdt_data.lim,0FFFFh     ;(1)Граница сегмента данных
        mov     gdt_code.lim,0FFFFh     ;(2)Граница сегмента команд
```

```
        mov   gdt_stack.lim,0FFFFh  ;(3)Граница сегмента стека
        mov   gdt_screen.lim,0FFFFh ;(4)Граница доп. сегмента
        push  DS                    ;(5)Загрузим теневой регистр
        pop   DS                    ;(6)сегмента данных
        push  SS                    ;(7)Загрузим теневой регистр
        pop   SS                    ;(8)сегмента стека
        push  ES                    ;(9)Загрузим теневой регистр
        pop   ES                    ;(10)дополнительного сегмента данных
;Выполним дальний переход для того, чтобы заново загрузить селектор
;в регистр CS и модифицировать его теневой регистр
        db 0EAh                     ;(11)Командой дальнего перехода
        dw offset go                ;(12)загрузим теневой регистр
        dw 16                       ;(13)сегмента команд
;Переключим режим процессора
go:     mov   EAX,CR0               ;(14)Получим содержимое регистра CR0
        and   EAX,0FFFFFFFEh ;(15)Сбросим бит защищенного режима
        mov   CR0,EAX               ;(16)Запишем назад в CR0
        db    0EAh                  ;(17)Код команды far jmp
        dw    offset return ;(18)Смещение
        dw    text                  ;(19)Сегмент
;---------------------------------------------------;
; Теперь процессор снова работает в реальном режиме ;
;---------------------------------------------------;
return:                     (20)
;Восстановим вычислительную среду реального режима
        mov   AX,data               ;(21)Сделаем адресуемыми данные
        mov   DS,AX                 ;(22)
        mov   AX,stk                ;(23)Сделаем адресуемым стек
        mov   SS,AX                 ;(24)
        sti                         ;(25)Разрешим аппаратные прерывания
;Работаем в DOS
        ...
```

В предложениях 1...4 приведенного фрагмента в поля границ всех четырех дескрипторов заносится FFFFh. Далее выполняется загрузка селекторов в сегментные регистры, что приводит к перезаписи содержимого теневых регистров. Загрузку селекторов можно выполнить любым доступным способом, например просто занося в них значения селекторов:

```
        mov   AX,8
        mov   DS,AX
```

и т. д. В примере 65.2 загрузка сегментных регистров осуществляется с использованием стека. Регистр CS загрузить непосредственно нельзя, поэтому его загрузку придется выполнить с помощью искусственно сформированной команды дальнего перехода (предложения 11...13).

Настроив все использовавшиеся в программе сегментные регистры, можно сбросить бит 0 в CR0. После перехода в реальный режим нам придется еще раз выполнить команду дальнего перехода (уже на точку входа блока работы в реальном режиме return), чтобы загрузить в регистр CS вместо хранящегося там селектора обычный сегментный адрес регистра команд (предложения 17...19).

Теперь процессор снова работает в реальном режиме, причем, хотя в сегментных регистрах DS, ES и SS остались незаконные для реального режима селекторы, программа будет какое-то время выполняться правильно, так как в теневых регистрах находятся правильные линейные адреса (оставшиеся от защищенного режима) и законные для реального режима границы (загруженные туда нами в предложениях 1...4).

Однако если в программе встретятся команды сохранения и восстановления содержимого сегментных регистров, например

```
        push   DS
        ...
        pop    DS
```

выполнение программы будет нарушено, так как команда pop DS загрузит в DS не сегментный адрес реального режима, а селектор, т. е. число 8 в нашем случае. Это число будет рассматриваться процессором как сегментный адрес, и дальнейшие обращения к полям данных приведут к адресации начиная с физического адреса 80h, что, конечно, лишено смысла. Даже если в нашей программе нет строк сохранения и восстановления сегментных регистров, они неминуемо встретятся, как только произойдет переход в DOS по команде int 21h, так как диспетчер DOS сохраняет, а затем восстанавливает все регистры задачи, в том числе и сегментные. Поэтому после перехода в реальный режим необходимо загрузить в используемые далее сегментные регистры соответствующие сегментные адреса, что и выполняется для регистров DS и SS в предложениях 21...24. Регистр ES мы не восстанавливаем, так как не будем им пользоваться.

В примере 65.1 после перехода в реальный режим потребовалось заново инициализировать указатель стека (см. предложение 87 примера 65.1), так как в процессе сброса процессора его содержимое разрушается. В новом варианте в этом нет необходимости: программная смена режима не влияет на содержимое регистров общего назначения.

После разрешения аппаратных прерываний возврат в реальный режим можно считать завершенным, и оставшаяся часть программы не отличается от примера 65.1.

Статья 66. **Работа с расширенной памятью**

В предыдущих примерах мы имели дело с сегментами небольшого размера (не более 64 Кбайт), размещаемыми в обычной памяти. В настоящей статье будет рассмотрена работа с большими сегментами данных, находящимися в расширенной памяти, т. е. за пределами первого мегабайта. Поскольку описание больших сегментов данных имеет некоторую специфику, мы начнем с более подробного, чем прежде, рассмотрения дескриптора памяти. Вообще дескрипторы могут быть трех типов:

- памяти;
- системные;
- шлюзы.

Форматы дескрипторов памяти и системных практически совпадают, формат же шлюза несколько отличается. Здесь мы рассмотрим только дескрипторы, служащие для описания сегментов памяти, т. е. тех сегментов, в которых располагаются команды, данные и стек программы. Формат дескриптора памяти (он соответствует структуре descr, которую мы использовали в примерах) изображен на рис. 66.1.

Как видно из рисунка, дескриптор занимает 8 байт. В байтах 2...4 и 7 записывается линейный сегментный адрес базы сегмента. Поскольку базовый адрес имеет длину 32 бита, он может быть назначен в любой точке 4-гигабайтового адресного пространства (естественно, только там, где имеется реальная оперативная память или другие адресуемые объекты, например видеопамять). В байтах 0-1 записываются младшие 16

бит границы сегмента, а в младшие 4 бита байта атрибутов 2 – оставшиеся биты 16...19. Таким образом, граница описывается 20 битами, и ее численное значение не может превышать 1 М.

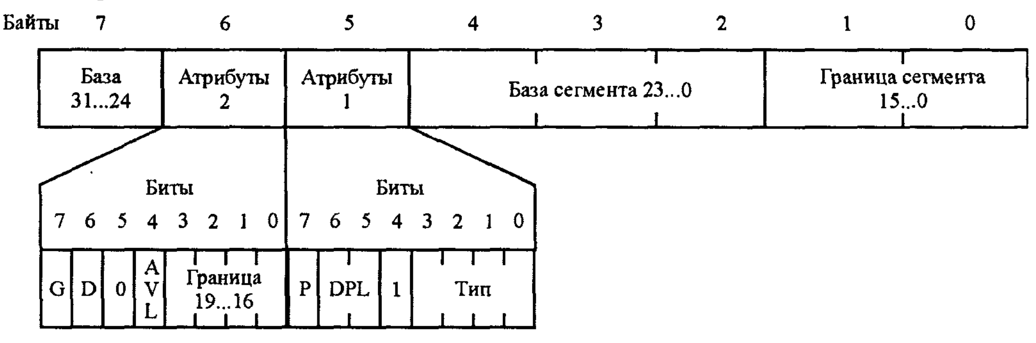


Рис. 66.1. Формат дескриптора сегмента памяти

Однако, как уже упоминалось ранее, единицы, в которых задается граница, можно изменять, что осуществляется с помощью бита дробности G (бит 7 байта атрибутов 2). Если G=0, граница указывается в байтах; если 1 – в блоках по 4 Кбайт. Таким образом, размер сегмента можно задавать с точностью до байта, но тогда он не может быть больше 1 Мбайт; если же установить G=1, то сегмент может достигать 4 Гбайт, однако его размер будет кратен 4 Кбайт. При этом база сегмента и в том и в другом случае задается с точностью до байта.

При выделении программе сегментов большого размера (как это предусмотрено в приведенном ниже примере) следует иметь в виду особенность вычисления процессором значения границы. Если G=1, истинная граница сегмента определяется следующим образом:

```
Граница сегмента = граница в дескрипторе * 4K + 4095
```

Таким образом, сегмент всегда простирается до конца последнего 4 килобайтового блока. Пусть, например, базовый адрес сегмента равен нулю, граница тоже равна нулю и бит дробности установлен. Тогда истинная граница сегмента равна

```
0*4K+4095=4095
```

т. е. сегмент будет занимать адреса 0...4095 и иметь размер 4 Кбайт. Если установить значение границы, равное единице, размер сегмента будет 8 Кбайт и т. д.

Рассмотрим теперь атрибуты сегмента, занимающие 2 байта дескриптора.

Тип сегмента занимает 4 бита и может иметь 16 значений, перечисленных в табл. 66.1.

Таблица 66.1. Значения поля типа дескриптора сегмента памяти

Тип	*Характеристики сегмента*
0h	Разрешено только чтение (сегмент данных)
1h	Разрешено только чтение, было обращение
2h	Разрешены чтение и запись (сегмент данных)
3h	Разрешены чтение и запись, было обращение
4h	Расширение вниз, разрешено только чтение (сегмент стека)
5h	Расширение вниз, разрешено только чтение, было обращение

6h	Расширение вниз, разрешены чтение и запись (сегмент стека)
7h	Расширение вниз, разрешены чтение и запись, было обращение
8h	Разрешено только исполнение (сегмент команд)
9h	Разрешено только исполнение, было обращение
0Ah	Разрешены исполнение и чтение (сегмент команд)
0Bh	Разрешены исполнение и чтение, было обращение
0Ch	Разрешено только исполнение, подчиненный сегмент
0Dh	Разрешено только исполнение, подчиненный сегмент, было обращение
0Eh	Разрешены исполнение и чтение, подчиненный сегмент
0Fh	Разрешены исполнение и чтение, подчиненный сегмент, было обращение

Как видно из этой таблицы, тип определяет правила доступа к сегменту. Указав для некоторого сегмента данных тип 0, мы защитим его от модификации; при попытке записи в такой сегмент возникнет исключение общей защиты. Сегменту команд можно присвоить тип 4, и тогда его можно только исполнять, однако система Windows обычно присваивает сегментам команд тип 0Ah. Это дает возможность программе обращаться к сегменту команд с целью чтения его содержимого.

Младший бит типа (его называют битом A – от Accessed, было обращение) устанавливается процессором в тот момент, когда в какой-либо сегментный регистр загружается селектор данного сегмента. Анализируя биты обращения различных сегментов, программа (в частности, операционная система защищенного режима) может судить о том, было ли обращение к данному сегменту после того, как она сбросила бит А.

Подчиненные, или согласованные (conforming), сегменты обычно используются для хранения подпрограмм общего пользования; для них не действуют общие правила защиты программ друг от друга.

Сегменты с расширением вниз, т. е. в сторону меньших адресов, используются для организации стеков, которые по ходу выполнения программы приходится расширять. Для относительно простых программ размеры сегментов обычно фиксированы и для стека можно использовать обычный сегмент данных (естественно, с разрешением как чтения, так и записи).

Бит 4 байта атрибутов 1 является идентификатором сегмента. Если он равен единице, как это показано на рис. 66.1, дескриптор описывает сегмент памяти. Значение этого бита 0 характеризует дескриптор системного сегмента.

Поле DPL (Descriptor Privilege Level, уровень привилегий дескриптора) служит для защиты программ друг от друга. Уровень привилегий может принимать значения от 0 (максимальные привилегии) до 3 (минимальные). Программам операционной системы обычно назначается уровень 0, прикладным программам – уровень 3, в результате чего исключается возможность некорректным программам разрушить операционную систему. В наших примерах операционная система защищенного режима отсутствует, программа сама выполняет функции операционной системы и всем ее сегментам назначается наивысший (нулевой) уровень привилегий, что открывает доступ ко всем средствам защищенного режима.

Бит P говорит о присутствии сегмента в памяти. В основном он используется в тех случаях, когда общий размер программы (или программ в многозадачном режиме) превышает объем наличной памяти и часть сегментов программ хранится на диске.

Тогда операционная система с помощью бита P определяет, находится ли требуемый сегмент в памяти, и при необходимости загружает его с диска. Перед выгрузкой ненужного сегмента на диск бит P сбрасывается. Для наших примеров естественно для всех сегментов иметь P=1.

Младшая половина байта атрибутов 2 занята старшими битами границы сегмента. Бит AVL (от Available, доступный) не используется и не анализируется процессором и предназначен для использования прикладными программами.

Бит D (Default, умолчание) определяет действующий по умолчанию размер для операндов и адресов. Он изменяет характеристики сегментов двух типов: исполняемых и стека. Если бит D сегмента команд равен нулю, в сегменте по умолчанию используются 16-битовые адреса и операнды, если единице – 32-битовые.

Атрибут сегмента, действующий по умолчанию, можно изменить на противоположный с помощью префиксов замены размера операнда (66h) и замены размера адреса (67h). Таким образом, для сегмента с D=0 префикс 66h перед некоторой командой заставляет ее рассматривать свои операнды как 32-битовые, а для сегмента с D=1 тот же префикс 66h, наоборот, сделает операнды 16-битовыми. В некоторых случаях транслятор сам включает в объектный модуль необходимые префиксы, в других случаях их приходится вводить в программу "вручную".

Поскольку мы запускаем наши программы в реальном режиме под управлением MS-DOS, естественно устанавливать D=0. Это, однако, не препятствует использованию в программе 32-битовых регистров (EAX, ECX и др.). Если транслятор встречается с командой, в качестве операнда которой используется 32-разрядный регистр, он включает в код команды префикс замены размера операнда 66h. Поэтому команды с явным обращением к 32-битовым операндам транслируются правильно. В то же время при использовании команды iret в защищенном режиме префикс 66h приходится включать в программу в явном виде, чтобы процессор, выполняя эту команду, снял со стека не три слова, как обычно, а три двойных слова. Без префикса команда iret будет выполняться неправильно.

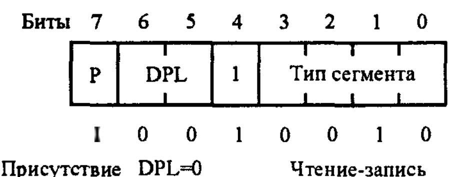
Префикс замены размера адреса 67h необходимо указывать, например, перед командой loop, если в качестве счетчика цикла используется не CX, а ECX. При отсутствии префикса команда loop (в сегменте, где по умолчанию используется 16-битовая адресация) будет выполнять цикл CX, а не ECX раз.

Сегменты стека, адресуемые через регистр SS, с помощью бита D определяют, какой регистр использовать в качестве указателя стека в командах push и pop. Если D=0, используется регистр SP, если D=1, то ESP.

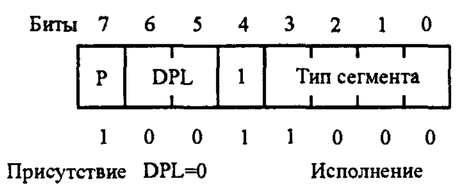
Сегменты команд 32-разрядных приложений транслируются с описателем размера адресов и операндов use32, а описывающие их дескрипторы должны иметь бит D=1.

Теперь мы можем расшифровать значения атрибутов, присвоенных нами сегментам наших программ (рис. 66.2).

Атрибут 1, сегменты данных и стека (92h)



Атрибут 1, сегмент команд (98h)



Атрибут 2, все сегменты (0)

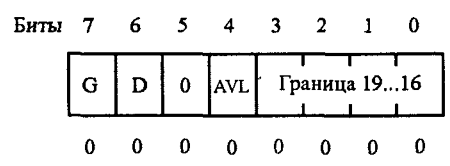


Рис. 66.2. Расшифровка значений атрибутов сегментов

В примере 66.1 в расширенной памяти создается сегмент размером 2 Мбайт, который заполняется натуральным рядом чисел (512 К 4-байтовых чисел) с визуальным контролем заполнения. За основу взята программа 65.1 (точнее, тот вариант, фрагмент которого приведен в примере 65.2). Помимо содержательных дополнений, связанных с обращением к расширенной памяти, в программу включены инструментальные средства диагностики и отладки – подпрограммы преобразования двоичного содержимого регистров и ячеек памяти в символьное представление, а также фрагмент вывода полученной диагностической строки на экран. Эти средства будут использоваться и в дальнейших примерах.

Пример 66.1. Заполнение 2 Мбайт расширенной памяти

```
.586P                       ;(1)
;Структура descr для описания дескрипторов сегментов
descr   struc               ;(2)Начало объявления структуры
lim     dw     0            ;(3)Граница (биты 0...15)
base_l  dw     0            ;(4)База, биты 0...15
base_m  db     0            ;(5)База, биты 16...23
attr_1  db     0            ;(6)Байт атрибутов 1
attr_2  db     0            ;(7)Граница (биты 16...19) и атрибуты 2
base_h  db     0            ;(8)База, биты 24...31
descr   ends                ;(9)Конец объявления структуры
;Сегмент данных
data    segment use16       ;(10)16-разрядный сегмент
;Таблица глобальных дескрипторов GDT
gdt_nulldescr <>            ;(11)Селектор 0
gdt_datadescr <data_size-1,,,92h>;(12)Селектор 8, сегмент данных
gdt_codedescr <code_size-1,,,98h>;(13)Селектор 16, сегмент команд
gdt_stack     descr <255,,,92h>;(14)Селектор 24, сегмент стека
gdt_screen descr <3999,8000h,0Bh,92h>;(15)Селектор 32, видеопамять
gdt_himem descr <511,0,10h,92h,80h>;(16)Селектор 40, старшая память
gdt_size=$-gdt_null         ;(17)Размер GDT
;Различные данные программы
pdescr  df     0            ;(18)Псевдодескриптор для команды lgdt
msg db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$';(19)
string db '**** ****-**** ****';(20)Шаблон диагностической строки
;          0    5    10   15   Позиции в шаблоне
```

```
len=$-string            ;(21)Длина строки
number  db '???? ????'  ;(22)Поле для динамического контроля
data_size=$-gdt_null    ;(23)Размер сегмента данных
data    ends            ;(24)Конец сегмента данных
;Сегмент команд
text    segment use16   ;(25)16-разрядный сегмент
        assume CS:text,DS:data;(26)
textseg label word      ;(27)Метка начала сегмента команд
main    proc            ;(28)Главная процедура
        xor   EAX,EAX   ;(29)Очистим EAX
        mov   AX,data   ;(30)Загрузим в DS сегментный адрес
        mov   DS,AX     ;(31)сегмента данных
;Вычислим и загрузим в GDT линейный адрес сегмента данных
        shl   EAX,4     ;(32)EAX=линейный базовый адрес
        mov   EBP,EAX   ;(33)Сохраним его в EBP для будущего
        mov   BX,offset gdt_data;(34)BX=смещение дескриптора
        mov   [BX].base_l,AX;(35)Загрузим младшую часть базы
        shr   EAX,16    ;(36)Старшую половину EAX в AX
        mov   [BX].base_m,AL;(37)Загрузим среднюю часть базы
;Вычислим и загрузим в GDT линейный адрес сегмента команд
        xor   EAX,EAX   ;(38)Очистим EAX
        mov   AX,CS     ;(39)Сегментный адрес сегмента команд
        shl   EAX,4     ;(40)
        mov   BX,offset gdt_code;(41)
        mov   [BX].base_l,AX;(42)
        shr   EAX,16    ;(43)
        mov   [BX].base_m,AL;(44)
;Вычислим и загрузим в GDT линейный адрес сегмента стека
        xor   EAX,EAX   ;(45)
        mov   AX,SS     ;(46)
        shl   EAX,4     ;(47)
        mov   BX,offset gdt_stack;(48)
        mov   [BX].base_l,AX;(49)
        shr   EAX,16    ;(50)
        mov   [BX].base_m,AL;(51)
;Подготовим псевдодескриптор pdescr для загрузки регистра GDTR
        mov   dword ptr pdescr+2,EBP;(52)База GDT
        mov   word ptr pdescr,gdt_size-1;(53)Граница GDT
        lgdt  pdescr    ;(54)Загрузим регистр GDTR
        cli             ;(55)Запрет прерываний
;Откроем линию A20 для обращения к расширенной памяти
        mov   AL,0D1h   ;(56)Команда управления
        out   64h,AL    ;(57)линией A20
        mov   AL,0DFh   ;(58)Код открытия
        out   60h,AL    ;(59)линии A20
;Переходим в защищенный режим
        mov   EAX,CR0   ;(60)Получим содержимое регистра CR0
        or    EAX,1     ;(61)Установим бит защищенного режима
        mov   CR0,EAX   ;(62)Запишем назад в CR0
;---------------------------------------------;
; Теперь процессор работает в защищенном режиме ;
;---------------------------------------------;
;Загружаем в CS:IP селектор:смещение точки continue
        db    0EAh      ;(63)Код команды far jmp
        dw    offset continue;(64)Смещение
        dw    16        ;(65)Селектор сегмента команд
continue:               ;(66)
;Делаем адресуемыми данные
        mov   AX,8      ;(67)Селектор сегмента данных
        mov   DS,AX     ;(68)
;Делаем адресуемым стек
```

```
        mov    AX,24        ;(69)Селектор сегмента стека
        mov    SS,AX        ;(70)
;Инициализируем ES
        mov    AX,32        ;(71)Селектор сегмента видеобуфера
        mov    ES,AX        ;(72)Инициализируем ES
;Заполним 2 Мбайт расширенной памяти
        mov    AX,40        ;(73)Селектор сегмента
        mov    GS,AX        ;(74)в расширенной памяти
        mov    EAX,0        ;(75)Первое число-заполнитель
        mov    EBX,0        ;(76)Индекс в сегменте
        mov    ECX,80000h   ;(77)80000h*4=200000h=2 Мбайт
fill:   mov    GS:[EBX],EAX;(78)Заполняем память!
;Выведем диагностическую строку о заполнении старшей памяти
        push   EAX          ;(79)Сохраним на время EAX
        push   CX           ;(80)и CX
        mov    SI,offset number+5;(81)Сюда младшую половину EAX
        call   wrd_asc      ;(82)Преобразуем в символы
        shr    EAX,16       ;(83)AX=старшая половина EAX
        mov    SI,offset number;(84)Сюда старшую половину EAX
        call   wrd_asc      ;(85)Преобразуем в символы
        mov    CX,9         ;(86)Длина строки
        mov    AH,43h       ;(87)Атрибут
        mov    DI,1760      ;(88)Начальная позиция на экране
scrnh:  lodsb               ;(89)Заберем из строки байт и
        stosw               ;(90)выведем в видеопамять слово
        loop   scrnh        ;(91)Цикл по строке
        pop    CX           ;(92)Восстановим CX
        pop    EAX          ;(93)и EAX
        add    EBX,4        ;(94)К следующему двойному слову памяти
        inc    EAX          ;(95)Следующее число-заполнитель
        db     67h          ;(96)Команда loop должна работать с ECX!
        loop   outc         ;(97)Переход на зацикливание
        jmp    ahead        ;(98)Выход из цикла после его окончания
outc:   jmp    fill         ;(99)Переход на начало цикла
;2 Мбайт заполнены. Проверим это прямым чтением 1-го и последнего слова
ahead:  mov    EBX,0        ;(100)Смещение 1-го слова
        mov    EAX,GS:[EBX];(101)Получим его
        mov    SI,offset string+5;(102)Куда заслать символы
        call   wrd_asc      ;(103)Преобразуем в символы
        shr    EAX,16       ;(104)Получим старшую часть слова
        mov    SI,offset string+0;(105)Куда заслать
        call   wrd_asc      ;(106)Преобразуем в символы
        mov    EBX,1FFFFCh  ;(107)Смещение последнего слова
        mov    EAX,GS:[EBX];(108)Получим его, далее аналогично
        mov    SI,offset string+15;(109)
        call   wrd_asc      ;(110)
        shr    EAX,16       ;(111)
        mov    SI,offset string+10;(112)
        call   wrd_asc      ;(113)
;Выведем на экран диагностическую строку
        mov    SI,offset string;(114)
        mov    CX,len       ;(115)
        mov    AH,74h       ;(116)
        mov    DI,1280      ;(117)
scrnl:  lodsb               ;(118)
        stosw               ;(119)
        loop   scrnl        ;(120)
;Закроем линию A20
        mov    AL,0D1h      ;(121)Команда управления
        out    64h,AL       ;(122)линией A20
        mov    AL,0DDh      ;(123)Код закрытия
```

```
        out     60h,AL       ;(124)линии A20
;Вернемся в реальный режим
;Сформируем и загрузим дескрипторы для реального режима
        mov     gdt_data.lim,0FFFFh;(125)Граница сегмента данных
        mov     gdt_code.lim,0FFFFh;(126)Граница сегмента команд
        mov     gdt_stack.lim,0FFFFh;(127)Граница сегмента стека
        mov     gdt_screen.lim,0FFFFh;(128)Граница доп. сегмента
        push    DS           ;(129)Загрузим теневой регистр
        pop     DS           ;(130)сегмента данных
        push    SS           ;(131)Загрузим теневой регистр
        pop     SS           ;(132)стека
        push    ES           ;(133)Загрузим теневой регистр
        pop     ES           ;(134)дополнительного сегмента
;Выполним дальний переход для того, чтобы заново загрузить селектор
;в регистр CS и модифицировать его теневой регистр
        db      0EAh         ;(135)Командой дальнего перехода
        dw      offset go    ;(136)загрузим теневой регистр
        dw      16           ;(137)сегмента команд
;Переключим режим процессора
go:     mov     EAX,CR0      ;(138)Получим содержимое регистра CR0
        and     EAX,0FFFFFFFEh;(139)Сбросим бит защищенного режима
        mov     CR0,EAX      ;(140)Запишем назад в CR0
        db      0EAh         ;(141)Код команды far jmp
        dw      offset return;(142)Смещение
        dw      text         ;(143)Сегмент
;-----------------------------------------------------;
; Теперь процессор снова работает в реальном режиме ;
;-----------------------------------------------------;
return:                      ;(144)
;Восстановим вычислительную среду реального режима
        mov     AX,data      ;(145)
        mov     DS,AX        ;(146)
        mov     AX,stk       ;(147)
        mov     SS,AX        ;(148)
        mov     SP,256       ;(149)
        sti                  ;(150)Разрешим аппаратные прерывания
        mov     AH,09h       ;(151)Проверим выполнение функций DOS
        mov     DX,offset msg;(152)после возврата в реальный режим
        int     21h          ;(153)
        mov     AX,4C00h     ;(154)Завершим программу обычным образом
        int     21h          ;(155)
main    endp                 ;(156)Конец главной процедуры
;Инструментальные средства - подпрограммы wrd_asc и bin_asc
;преобразования двоичного числа в символьное 16-ричное представление
;Подпрограмма wrd_asc преобразования слова
;При вызове преобразуемое число в AX, DS:SI → поле для результата
wrd_asc proc                 ;(157)
        pusha                ;(158)Сохраним все регистры
        mov     BX,0F000h    ;(159)В BX будет маска битов
        mov     DL,12        ;(160)В DL будет число сдвигов AX
        mov     CX,4         ;(161)Счетчик цикла
cccc:   push    CX           ;(162)Сохраним его
        push    AX           ;(163)Сохраним исходное число в стеке
        and     AX,BX        ;(164)Выделим четверку битов
        mov     CL,DL        ;(165)Отправим в CL число сдвигов
        shr     AX,CL        ;(166)Сдвинем на CL бит вправо
        call    bin_asc      ;(167)Преобразуем в символ ASCII
        mov     byte ptr [SI],AL;(168)Отправим в строку результата
        inc     SI           ;(169)Сдвинемся по строке вправо
        pop     AX           ;(170)Вернем в AX исходное число
```

```
        shr    BX,4          ;(171)Модифицируем маску битов
        sub    DL,4          ;(172)Модифицируем число сдвигов
        pop    CX            ;(173)Восстановим счетчик цикла
        loop   cccc          ;(174)Цикл
        popa                 ;(175)Восстановим все регистры
        ret                  ;(176)Возврат из подпрограммы
wrd_asc endp                 ;(177)
;Подпрограмма преобразования 16-ричной цифры
;Преобразуемая четверка битов в младшей половине AL, результат в AL
bin_asc proc                 ;(178)
        cmp    AL,9          ;(179)Цифра > 9?
        ja     lettr         ;(180)Да, на преобразование в букву
        add    AL,30h        ;(181)Нет, преобразуем в символ 0...9
        jmp    ok            ;(182)И на выход из подпрограммы
lettr:  add    AL,37h        ;(183)Преобразуем в символ A...F
ok:     ret                  ;(184)Возврат в вызывающую процедуру
bin_asc endp                 ;(185)
code_size=$-textseg          ;(186)Размер сегмента команд
text    ends                 ;(187)Конец сегмента команд
;Сегмент стека
stk     segment use16 stack;(188)16-разрядный сегмент
        db     256 dup ('^');(189)
stk     ends                 ;(190)
        end main             ;(191)
```

Желая образовать дополнительный сегмент объемом 2 Мбайт в расширенной памяти, мы должны прежде всего предусмотреть описывающий его дескриптор в таблице глобальных дескрипторов:

```
gdt_himem descr <511,0,10h,92h,80h,>;(16)Селектор 40
```

Будучи расположен в таблице GDT на шестом месте, этот дескриптор получает индекс 5, что соответствует селектору 40.

Линейный адрес первого байта расширенной памяти равен 100000h. Из этого адреса младшие 16 бит (0000h) записываются в поле base_l, следующие 8 бит, содержащие 10h, – kk в поле base_m и старшие 8 бит (00h) – в поле base_h.

Поскольку размер сегмента превышает 1 Мбайт, его границу следует указывать в блоках по 4 Кбайт. Поэтому мы устанавливаем в 1 бит дробности G в байте атрибутов 2, который, таким образом, принимает значение 80h. Размер сегмента (2 Мбайт) составляет 512 блоков по 4 Кбайт, поэтому значение границы составляет 511. Байт атрибутов 1, как и для других сегментов памяти, принимает значение 92h (для чтения и записи, присутствует).

В поля данных программы включены две символьные строки для диагностики и контроля хода выполнения программы. В строку number по мере заполнения расширенной памяти числами будет выводиться текущее число. Это даст возможность контролировать ход заполнения памяти и заодно проверить правильность содержимого последней заполненной ячейки. Строка string представляет собой шаблон для вывода на экран последовательности чисел, отображающих содержимое регистров или ячеек памяти. В процессе отладки и исследования программы в эту строку можно помещать любые данные, характеризующие операционную среду в тех или иных точках программы. Вывод перед завершением программы строки string на экран позволяет проанализировать ход выполнения программы. В данном примере строка string рассчитана на вывод двух длинных (двухсловных) чиссл – содержимого первой и последней ячейки заполняемого блока расширенной памяти; в последующих примерах объем ди-

агностической информации будет больше и длина строки string будёт соответственно увеличиваться.

Для преобразования двоичных чисел в символьную форму в программу включены подпрограммы wrd_asc и bin_asc, рассмотренные в статье 13. Подпрограмма wrd_asc преобразует в символьную форму число, содержащееся в регистре AX, помещая результат преобразования (4 символа) по адресу DS:SI. Преобразование содержимого расширенного регистра EAX приходится выполнять в два этапа:

```
        mov   EAX,данное
        mov   SI,offset string+5;Смещение в string младшей части числа
        call  wrd_asc       ;Преобразуем в символы
        shr   EAX,16        ;Сдвинем старшую часть слова в AX
        mov   SI,offset string+0;Смещение в string старшей части числа
                            ;(левее младшей)
        call   wrd_asc      ;Преобразуем в символы
```

Перед завершением программы заполненная предварительно строка string выводится на экран переносом ее в видеопамять:

```
;Выведем на экран диагностическую строку
        mov   SI,offset string;DS:SI → string
        mov   CX,len        ;Число байтов в string
        mov   AH,74h        ;Атрибут символа
        mov   DI,1280       ;Смещение на экране
scrnl:  lodsb               ;AL=очередной байт строки, AH=атрибут символа
        stosw               ;Перенос AX в видеопамять
        loop  scrnl         ;Цикл по длине строки
```

Перед переходом в защищенный режим (или после перехода в него) следует открыть линию A20, т. е. адресную линию, на которой устанавливается единичный уровень сигнала, если происходит обращение к мегабайтам адресного пространства с номерами 1, 3, 5 и т. д. (первый мегабайт имеет номер 0). В реальном режиме линия A20 заблокирована и, если значение адреса выходит за пределы FFFFFh, выполняется его циклическое оборачивание (линейный адрес 100000h превращается в 00000h, адрес 100001h – в 00001h и т. д.). Открытие (разблокирование) линии A20 выключает механизм циклического оборачивания адреса, что позволяет адресоваться к расширенной памяти. Управление блокированием линии A20 осуществляется через порт 64h, куда сначала следует послать команду D1h управления линией A20, а затем – код открытия DFh (предложения 56...59).

Переход в защищенный режим и действия по инициализации сегментных регистров выполняются обычным образом.

Заполнение расширенной памяти начинается с предложения 73. В свободный сегментный регистр GS заносится селектор сегмента в расширенной памяти и инициализируются регистры EAX, EBX и ECX (и число-заполнитель, и число шагов в цикле, и смещение в памяти превышают 64 К, поэтому требуются 32-разрядные регистры). В предложении 78 число-заполнитель отправляется в расширенную память. Для динамического контроля хода заполнения памяти в каждом шаге цикла очередное число выводится на экран (предложения 79...91). Далее выполняются инкременты индекса и числа-заполнителя и возврат в начало цикла.

Для того чтобы команда loop использовала в своей работе регистр ECX, а не CX (вспомним, что мы установили для сегмента команд D=0 и сделали тем самым программу 16-разрядной), перед ней в программу включен код префикса замены размера адреса (предложение 96). Другая сложность возникла из-за того, что в цикле оказалось

больше 128 байт, а команда loop может осуществлять только короткие переходы в диапазоне +127...–128 байт. Поэтому командой loop (предложение 97) выполняется переход не на начало цикла, а на вспомогательное предложение outc, в котором командой безусловного перехода управление может быть передано уже в любую точку программы. Команда безусловного перехода jmp в предложении 98 позволяет обойти предложение 99 после завершения цикла.

В программе предусмотрено контрольное чтение расширенной памяти и вывод в диагностическую строку string прочитанных чисел. В предложениях 100...106 читается и выводится первое 4-байтовое слово из сегмента расширенной памяти (которое должно быть равно 0000 0000h), в предложениях 107...113 – последнее 4-байтовое слово 2-мегабайтового сегмента (0007 FFFFh). Эта проверка может оказаться весьма полезной. Действительно, процессор не реагирует на отсутствие физической памяти по указанному в команде адресу. Поэтому если попытаться заполнить больше памяти, чем есть в компьютере (предварительно установив соответственно границу сегмента в дескрипторе), то программа будет работать так, как если бы эта память имелась, хотя, конечно, реально числа записываться в отсутствующую память не будут. Чтение из памяти последнего записанного в нее числа позволит убедиться в том, что это число действительно записано в память. Участок экрана с выводом программы показан на рис. 66.3.

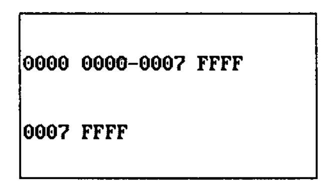


Рис. 66.3. Вид диагностических сообщений после завершения программы

Перед переходом в реальный режим следует закрыть линию A20, для чего в порт 64h посылается команда D1h управления линией A20, а затем – код закрытия линии DDh (предложения 121...124). Завершенис программы выполняется, как и ранее.

Читатель может модифицировать программу, увеличив или уменьшив размер сегмента в расширенной памяти и, соответственно, число шагов в цикле заполнения, настроив ее под конкретную конфигурацию своего компьютера.

Статья 67. **Исключения**

Наши первые программы защищенного режима, рассмотренные в предыдущих статьях, работали в условиях запрещенных аппаратных прерываний. К программным прерываниям, реализуемым с помощью команды int, мы также не обращались. Все эти предосторожности были необходимы потому, что механизм обработки прерываний в защищенном режиме сильно отличается от механизма реального режима. Если бы мы, не активизируя механизм обработки прерываний защищенного режима, перешли в защищенный режим при разрешенных прерываниях, первое же аппаратное прерывание (например, от таймера) привело бы к отключению процессора. То же произошло бы при выполнении процессором команды int с любым номером. Между прочим, попытка проверить это утверждение с помощью примера 65.1 не приведет к успеху. Действительно, в этой программе возврат в реальный режим осуществляется как раз с помощью сброса процессора, причем мы заранее предусмотрительно настроили КМОП-

микросхему и ячейки области данных BIOS, чтобы после отключения процессор вернулся в нашу программу. Поэтому в такой программе нельзя обнаружить ошибки защищенного режима, приводящие к отключению процессора.

Иное дело программы 65.2 или 66.1. Здесь возврат в реальный режим осуществляется программным способом, сбросом бита 0 управляющего регистра CR0. Если ошибка в программе имеет своим следствием отключение процессора, то программа аварийно завершится и будет инициирована перезагрузка системы. Происходит это потому, что байт состояния отключения в КМОП-микросхеме, расположенный по адресу Fh, в исходном состоянии содержит код 0. Это значение кода приводит после сброса процессора (и после нажатия клавиш Ctrl+Alt+Del) к перезапуску системы. Включите в программу (в ту ее часть, которая выполняется в защищенном режиме) команду int с любым числовым аргументом (например, int 3 или int 21h). Компьютер перезагрузится.

В этой и последующих статьях будет рассмотрена система прерываний защищенного режима и описаны общие принципы обработки как аппаратных, так и программных прерываний.

Как уже упоминалось в статье 25, в МП 86 (и соответственно, в реальном режиме 32-разрядных процессоров) предусмотрены прерывания трех видов: внутренние, возникающие в самом микропроцессоре, внешние, поступающие в процессор от внешних устройств компьютера через контроллеры прерываний, и программные, инициируемые командой int. В защищенном режиме также возможны прерывания всех трех типов, при этом число внешних прерываний, определяемое количеством контроллеров прерываний в компьютере, осталось, естественно, тем же, однако функции внутренних прерываний и их количество существенно расширены. Внутренние прерывания, называемые здесь исключениями, исключительными ситуациями или особыми случаями (exceptions), являются важнейшим элементом организации защищенного режима. Мы начнем знакомство с системой прерываний защищенного режима именно с исключений. Внешние, а также программные прерывания будут рассмотрены в последующих статьях, хотя принцип обслуживания всех видов прерываний один и многие рассматриваемые в настоящей статье детали будут относиться ко все трем видам прерываний.

Так же как и в реальном режиме, все прерывания защищенного режима имеют свои номера, причем их общее количество не должно превышать 256. Под исключения отданы первые 32 номера (0...31), хотя реально возникающие исключения имеют номера 0...17, а номера с 18 по 31 зарезервированы для будущих моделей процессоров.

В реальном режиме процессор при регистрации прерывания обращается к таблице векторов прерываний, находящейся всегда в самом начале памяти и содержащей двухсловные адреса программ обработки прерываний, обычно называемых обработчиками прерываний. В защищенном режиме аналогом таблицы векторов прерываний является таблица дескрипторов прерываний – IDT (Interrupt Descriptor Table), располагающаяся обычно в операционной системе защищенного режима. Таблица IDT содержит дескрипторы обработчиков прерываний, в которые, в частности, входят их адреса. Для того чтобы процессор мог обратиться к этой таблице, ее адрес следует загрузить в регистр IDTR (Interrupt Descriptor Table Register, регистр таблицы дескрипторов прерываний) в точности так же, как это делается с адресом таблицы глобальных дескрипторов GDT.

Если в таблице глобальных дескрипторов GDT собраны дескрипторы сегментов программы, то таблица дескрипторов прерываний IDT состоит из дескрипторов другого вида, которые называются шлюзами (вентилями). Через шлюзы осуществляется доступ к обработчикам прерываний и исключений. Формат шлюза заметно отличается от формата дескриптора сегмента памяти, причем ясно, что в нем должен присутствовать адрес обработчика соответствующего прерывания. Процессор, зарегистрировав прерывание или исключение, по его номеру извлекает из IDT шлюз, определяет адрес обработчика и передает ему управление. Обработчик должен заканчиваться командой iret, которая возвращает управление в прерванную программу.

Формат шлюза (вместе с обозначениями полей, которые будут использованы в примере 67.1) изображен на рис. 67.1.

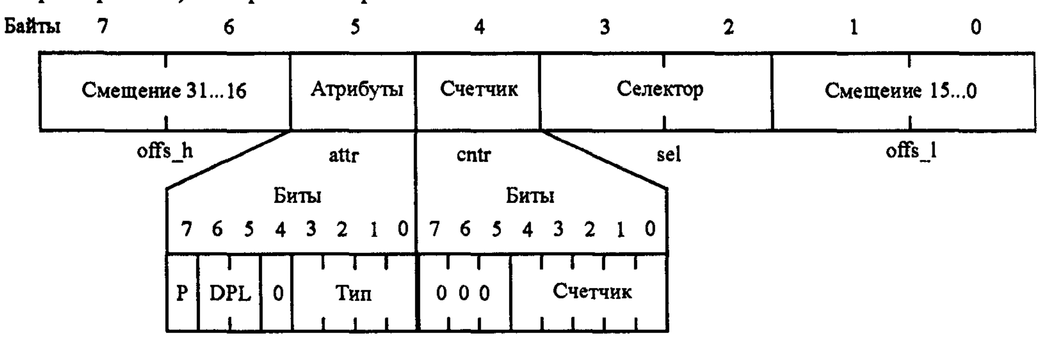


Рис. 67.1. Формат шлюзов, входящих в таблицу дескрипторов прерываний

Если основной частью содержимого сегмента памяти был его линейный адрес, то в шлюзе указывается полный трехсловный адрес обработчика, состоящий из селектора и смещения. Смещение имеет 32 бит и занимает в дескрипторе два поля – байты 0...1 и 6...7. Селектор имеет 16 бит и занимает байты 2...3.

Байт атрибутов имеет такую же структуру, как и в дескрипторах памяти, и включает тип, идентификатор дескриптора шлюза (бит 4), равный нулю, уровень привилегий дескриптора DPL и бит присутствия P. Тип дескриптора может принимать значения, перечисленные в табл. 67.1.

Таблица 67.1. Значения поля типа для дескрипторов шлюзов

Тип	*Назначение дескриптора*
4	Шлюз вызова 80286
5	Шлюз задачи
6	Шлюз прерываний 80286
7	Шлюз ловушки 80286
Ch	Шлюз вызова 80386, i486, Pentium
Eh	Шлюз прерываний 80386, i486, Pentium
Fh	Шлюз ловушки 80386, i486, Pentium

Поле счетчика предназначено для хранения числа параметров (каждый размером в двойное слово), копируемых с одного стека на другой в тех случаях, когда использование шлюза приводит к переходу на другой уровень привилегий и, соответственно, на другой стек (стек другого уровня). Понятие уровней привилегий и проблемы перехода

с уровня на уровень будут рассмотрены в статье 72; в примере настоящей статьи все составляющие программного комплекса принадлежат уровню 0 и вопроса о передаче параметров не возникает.

Итак, в программе, обслуживающей исключения, следует сформировать таблицу дескрипторов прерываний IDT, поместить в нее адреса обработчиков исключений, предусмотренных в программе, загрузить адрес IDT в системный регистр процессора IDTR и перейти в защищенный режим. Пока процессор находится в защищенном режиме, он при возникновении исключений (или вообще прерываний) будет использовать таблицу дескрипторов прерываний; после возврата в реальный режим ее следует заменить таблицей векторов реального режима. Рассмотрим простейшую программу, в которой предусмотрены средства обработки исключений (хотя сама обработка сведена к минимуму). Для облегчения отладки программы и исследования ее работы в нее включены рассмотренные в статье 13 и использованные уже в статье 76 подпрограммы wrd_asc и bin_asc, с помощью которых можно преобразовывать в символьную форму, с целью последующего вывода на экран, содержимое регистров или, если потребуется, ячеек памяти.

Пример 67.1, как все последующие, во многом повторяет рассмотренные выше программы. Такие программные блоки, как вычисление линейных адресов сегментов и заполнение таблицы глобальных дескрипторов, переход в защищенный и возврат в реальный режим, вывод на экран диагностических сообщений из защищенного и реального режимов, и прочие в большинстве наших примеров останутся практически без изменений. В будущем мы не будем повторять в текстах примеров эти программные блоки, ограничиваясь указанием заголовочных комментариев. Однако программа примера 67.1, учитывая ее относительную сложность, приведена целиком, за исключением текстов процедур wrd_asc и bin_asc, которые уже были рассмотрены в предыдущем примере.

Пример 67.1. Исключения

```
.586P                      ;(1)
;Структура для описания дескрипторов сегментов
descr   struc              ;(2)
lim     dw 0               ;(3)Граница (биты 0...15)
base_l  dw 0               ;(4)База, биты 0...15
base_m  db 0               ;(5)База, биты 16...23
attr_1  db 0               ;(6)Байт атрибутов 1
attr_2  db 0               ;(7)Граница (биты 16...19) и атрибуты 2
base_h  db 0               ;(8)База, биты 24...31
descr   ends               ;(9)
;Структура для описания шлюзов ловушек
trap    struc              ;(10)
offs_l  dw 0               ;(11)Смещение обработчика, биты 0...15
sel     dw 16              ;(12)Селектор сегмента команд
cntr    db 0               ;(13)Не используется
dtype   db 8Fh             ;(14)Тип шлюза - ловушка 80386 и выше
offs_h  dw 0               ;(15)Смещение обработчика, биты 16...31
trap    ends               ;(16)
;Сегмент данных
data    segment use16      ;(17)16-разрядный сегмент
;Таблица глобальных дескрипторов GDT
gdt_null descr <>          ;(18)Селектор 0
gdt_data descr <data_size-1,,,92h>;(19)Селектор 8, сегмент данных
gdt_code descr <code_size-1,,,98h>;(20)Селектор 16, сегмент команд
gdt_stack descr <255,,,92h,,>;(21)Селектор 24, сегмент стека
```

```
gdt_screen descr <3999,8000h,0Bh,92h> ;(22)Селектор 32, видеопамять
gdt_size=$-gdt_null        ;(23)Размер GDT
idt label word             ;(24)Метка начала таблицы дескрипторов исключений
trap <exc0>                ;(25)Дескриптор исключения 0
trap <dummy>               ;(26)Дескриптор исключения 1
trap <dummy>               ;(27)Дескриптор исключения 2
trap <exc3>                ;(28)Дескриптор исключения 3
trap <dummy>               ;(29)Дескриптор исключения 4
trap <dummy>               ;(30)Дескриптор исключения 5
trap <dummy>               ;(31)Дескриптор исключения 6
trap <dummy>               ;(32)Дескриптор исключения 7
trap <dummy>               ;(33)Дескриптор исключения 8
trap <dummy>               ;(34)Дескриптор исключения 9
trap <exc10>               ;(35)Дескриптор исключения 10
trap <exc11>               ;(36)Дескриптор исключения 11
trap <exc12>               ;(37)Дескриптор исключения 12
trap <exc13>               ;(38)Дескриптор исключения 13
trap <dummy>               ;(39)Дескриптор исключения 14
trap <dummy>               ;(40)Дескриптор исключения 16
trap <dummy>               ;(41)Дескриптор исключения 17
trap <dummy>               ;(42)Дескриптор исключения 18
idt_size=$-idt             ;(43)Размер таблицы дескрипторов исключений
;Различные данные программы
pdescr  df 0               ;(44)Псевдодескриптор для команд lgdt и lidt
sym     db 1               ;(45)Символ для вывода на экран
attr    db 1Eh             ;(46)Его атрибут
msg db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$';(47)
string  db '**** ****-**** ****-**** ****';(48)
;            0    5    10   15   20   25   Позиции в шаблоне
len=$-string               ;(49)
data_size=$-gdt_null       ;(50)Размер сегмента данных
data    ends               ;(51)
;Сегмент команд
text    segment use16      ;(52)16-разрядный сегмент
        assume CS:text,DS:data;(53)
textseg label word         ;(54)Метка начала сегмента команд
exc0    proc               ;(55)Обработчик исключения 0
        mov   AX,0         ;(56)Для вывода на экран номера исключения
        jmp   home         ;(57)На выход
exc0    endp               ;(58)
exc3    proc               ;(59)Обработчик исключения 3
        mov   AX,3         ;(60)Для вывода на экран номера исключения
        jmp   home         ;(61)На выход
exc3    endp               ;(62)
exc10   proc               ;(63)Обработчик исключения 10
        mov   AX,0Ah       ;(64)Для вывода на экран номера исключения
        jmp   home         ;(65)На выход
exc10   endp               ;(66)
exc11   proc               ;(67)Обработчик исключения 11
        mov   AX,0Bh       ;(68)Для вывода на экран номера исключения
        jmp   home         ;(69)На выход
exc11   endp               ;(70)
exc12   proc               ;(71)Обработчик исключения 12
        mov   AX,0Ch       ;(72)Для вывода на экран номера исключения
        jmp   home         ;(73)На выход
exc12   endp               ;(74)
exc13   proc               ;(75)Обработчик исключения 13
        mov   AX,0Dh       ;(76)Для вывода на экран номера исключения
        jmp   home         ;(77)На выход
exc13   endp               ;(78)
dummy   proc               ;(79)Обработчик остальных исключений
```

```
        mov     AX,5555h      ;(80)Условный код остальных исключений
        jmp     home          ;(81)На выход
dummy   endp                  ;(82)
main    proc                  ;(83)
        xor     EAX,EAX       ;(84)Очистим EAX
        mov     AX,data       ;(85)Загрузим в DS сегментный
        mov     DS,AX         ;(86)адрес сегмента данных
;Вычислим и загрузим в GDT линейный адрес сегмента данных
        shl     EAX,4         ;(87)EAX=линейный базовый адрес
        mov     EBP,EAX       ;(88)Сохраним его в EBP для будущего
        mov     BX,offset gdt_data;(89)BX=смещение дескриптора
        mov     [BX].base_l,AX;(90)Загрузим младшую часть базы
        shr     EAX,16        ;(91)Старшую половину EAX в AX
        mov     [BX].base_m,AL;(92)Загрузим среднюю часть базы
;Вычислим и загрузим в GDT линейный адрес сегмента команд
        xor     EAX,EAX       ;(93)Очистим EAX
        mov     AX,CS         ;(94)Сегментный адрес сегмента команд
        shl     EAX,4         ;(95)
        mov     BX,offset gdt_code;(96)
        mov     [BX].base_l,AX;(97)
        shr     EAX,16        ;(98)
        mov     [BX].base_m,AL ;(99)
;Вычислим и загрузим в GDT линейный адрес сегмента стека
        xor     EAX,EAX       ;(100)
        mov     AX,SS         ;(101)
        shl     EAX,4         ;(102)
        mov     BX,offset gdt_stack ;(103)
        mov     [BX].base_l,AX;(104)
        shr     EAX,16        ;(105)
        mov     [BX].base_m,AL;(106)
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        mov     dword ptr pdescr+2,EBP;(107)База GDT
        mov     word ptr pdescr,gdt_size-1;(108)Граница GDT
        lgdt    pdescr        ;(109)Загрузим регистр GDTR
        cli                   ;(110)Запрет аппаратных прерываний
;Загрузим IDTR
        mov     word ptr pdescr,idt_size-1 ;(111)Граница
        xor     EAX,EAX       ;(112)
        mov     AX,offset idt;(113)
        add     EAX,EBP       ;(114)Линейный адрес IDT
        mov     dword ptr pdescr+2,EAX;(115)В псевдодескриптор
        lidt    pdescr        ;(116)Загрузка IDTR
;Переходим в защищенный режим
        mov     EAX,CR0       ;(117)Получим содержимое регистра CR0
        or      EAX,1         ;(118)Установим бит защищенного режима
        mov     CR0,EAX       ;(119)Запишем назад в CR0
;---------------------------------------------;
; Теперь процессор работает в защищенном режиме ;
;---------------------------------------------;
;Загружаем в CS:IP селектор:смещение точки continue
        db 0EAh               ;(120)Код команды far jmp
        dw offset continue;(121)Смещение
        dw 16                 ;(122)Селектор сегмента команд
continue:                     ;(123)
;Делаем адресуемыми данные
        mov     AX,8          ;(124)Селектор сегмента данных
        mov     DS,AX         ;(125)
;Делаем адресуемым стек
        mov     AX,24         ;(126)Селектор сегмента стека
        mov     SS,AX         ;(127)
;Инициализируем ES
```

```
        mov     AX,32         ;(128)Селектор сегмента видеобуфера
        mov     ES,AX         ;(129)Инициализируем ES
;Выводим на экран тестовую строку символов
        mov     DI,1920       ;(130)Начальная позиция на экране
        mov     CX,80         ;(131)Число выводимых символов
        mov     AX,word ptr sym;(132)Символ+атрибут
scrn:   stosw                 ;(133)Содержимое AX на экран
        inc     AL            ;(134)Инкремент символа
        loop    scrn          ;(135)Цикл
        mov     AX,0FFFFh     ;(136)Условный код нормального завершения
home:   mov     SI,offset string;(137)Точка перехода из обработчиков
        call    wrd_asc       ;(138)Преобразование AX в символьную строку
;Выведем на экран диагностическую строку
        mov     SI,offset string;(139)
        mov     CX,len        ;(140)
        mov     AH,74h        ;(141)
        mov     DI,1280       ;(142)
scrn1:  lodsb                 ;(143)
        stosw                 ;(144)
        loop    scrn1         ;(145)
;Вернемся в реальный режим
;Сформируем и загрузим дескрипторы для реального режима
        mov     gdt_data.lim,0FFFFh;(146)Граница сегмента данных
        mov     gdt_code.lim,0FFFFh;(147)Граница сегмента команд
        mov     gdt_stack.lim,0FFFFh;(148)Граница сегмента стека
        mov     gdt_screen.lim,0FFFFh;(149)Граница доп. сегмента
        push    DS            ;(150)Загрузим теневой регистр
        pop     DS            ;(151)сегмента данных
        push    SS            ;(152)Загрузим теневой регистр
        pop     SS            ;(153)стека
        push    ES            ;(154)Загрузим теневой регистр
        pop     ES            ;(155)дополнительного сегмента
;Выполним дальний переход для того, чтобы заново загрузить селектор
;в регистр CS и модифицировать его теневой регистр
        db      0EAh          ;(156)Командой дальнего перехода
        dw      offset go     ;(157)загрузим теневой регистр
        dw      16            ;(158)сегмента команд
;Переключим режим процессора
go:     mov     EAX,CR0       ;(159)Получим содержимое регистра CR0
        and     EAX,0FFFFFFFEh;(160)Сбросим бит защищенного режима
        mov     CR0,EAX       ;(161)Запишем назад в CR0
        db      0EAh          ;(162)Код команды far jmp
        dw      offset return;(163)Смещение
        dw      text          ;(164)Сегмент
;-----------------------------------------------------;
; Теперь процессор снова работает в реальном режиме ;
;-----------------------------------------------------;
return:                       ;(165)
;Восстановим вычислительную среду реального режима
        mov     AX,data       ;(166)Сделаем адресуемыми данные
        mov     DS,AX         ;(167)
        mov     AX,stk        ;(168)Сделаем адресуемым стек
        mov     SS,AX         ;(169)
        mov     SP,256        ;(170)Настроим SP
;Восстановим состояние регистра IDTR реального режима
        mov     AX,3FFh       ;(171)Граница таблицы векторов (1 Кбайт)
        mov     word ptr pdescr,AX;(172)
        mov     EAX,0         ;(173)Смещение таблицы векторов
        mov     dword ptr pdescr+2,EAX;(174)
        lidt    pdescr        ;(175)Загрузим псевдодескриптор в IDTR
        sti                   ;(176)Разрешим аппаратные прерывания
```

```
;Работаем в DOS
        mov   AH,09h        ;(177)Проверим выполнение функций DOS
        mov   DX,offset string;(178)после возврата в реальный режим
        int   21h           ;(179)
        mov   AX,4C00h      ;(180)Завершим программу обычным образом
        int   21h           ;(181)
main    endp                ;(182)
;Инструментальные средства - подпрограммы wrd_asc и bin_asc
;преобразования двоичного числа в символьное 16-ричное представление
wrd_asc proc                ;(183)
        ...
wrd_asc endp                ;(184)
bin_asc proc                ;(185)
        ...
bin_asc endp                ;(186)
code_size=$-textseg         ;(187)Размер сегмента команд
text    ends                ;(188)Конец сегмента команд
;Сегмент стека
stk     segment stack use16;(189)16-разрядный сегмент
        db 256 dup ('^')    ;(190)
stk     ends                ;(191)Конец сегмента стека
        end main            ;(192)Конец программы
```

Программа 67.1 имеет в целом ту же структуру, что и предыдущие. Она начинается с описания структуры глобального дескриптора (предложсния 2...9), за которым следуст структура дескриптора прерывания (предложения 10...16). В таблицу IDT могут входить шлюзы следующих типов (см. табл. 67.1):

- шлюз задачи;
- шлюз прерывания МП 286;
- шлюз прерывания МП 386, 486 и Pentium;
- шлюз ловушки МП 286;
- шлюз ловушки МП 386, 486 и Pentium.

Через шлюзы задачи (task) осуществляется переключение на другие задачи в многозадачном режиме; через шлюзы прерываний (interrupt) обслуживаются аппаратные прерывания; шлюзы ловушск (trap) служат для обработки исключений и программных прерываний. В данной программе используются шлюзы ловушек, для которых байт атрибута принимаст значение 8Fh (присутствие, DPL=0, шлюз ловушки 80386 и выше.

Сегмент данных начинается, как и в предыдущих примерах, с таблицы глобальных дескрипторов. За ней описана таблица дескрипторов прерываний IDT (предложения 24...43), в которую пока включены лишь шлюзы исключений, а шлюзы аппаратных прерываний отсутствуют. Поскольку в процсссоре в принципе могут возникать исключсния с номерами 0...17, в IDT включены 18 дескрипторов, каждый из которых содержит смсщение соответствующего обработчика. Однако не все исключения можно ожидать в наших программах. Например, вряд ли мы столкнемся с исключением 5 – нарушения границ массива или исключением 7 – отсутствия сопроцессора. Для упрощения программы в нее включены лишь обработчики наиболее всроятных исключений: 0 – ошибки деления, 3 – команды int 3, 10 – недопустимого сегмента состояния задачи, 11 – отсутствия ссгмента, 12 – ошибки обращения к стеку и 13 – общей защиты (подробнсе о видах исключений и причинах их возникновения будет рассказано в следующей статье). Смещения этих обработчиков (exc0, exc3 и др.) указаны в соотвстст-

вующих по номеру макросах trap. Для остальных дескрипторов предусмотрен единый обработчик с именем dummy.

Индивидуальной настройке в дескрипторах исключений подлежат лишь адреса обработчиков. Остальные поля для всех дескрипторов одинаковы и описаны в структуре trap.

Следует заметить, что в нашей программе дескрипторы исключений полностью заполняются в процессе трансляции и в дальнейшем тексте программы обращаться к этим полям данных уже не нужно. В этом случае дескрипторы могут не иметь мнемонических имен (как это сделано для дескрипторов GDT); важно только, чтобы они располагались в таблице IDT в правильном порядке. Константе idt_size транслятор присваивает значение длины таблицы IDT, которая потребуется при загрузке регистра IDTR.

В сегмент данных включена строка string, которая совместно с подпрограммами word_asc и bin_asc будет использоваться для формирования и вывода на экран диагностического сообщения.

Сегмент команд начинается с процедур обработчиков исключений. Все они имеют одинаковую структуру: в регистр AX засылается номер данного исключения, после чего осуществляется переход в точку home. Следует отметить, что это не очень изящный метод завершения обработки исключения. Как уже отмечалось, возврат из обработчика исключения обычно осуществляется командой iret, которая выполняет переход либо на ту команду, при выполнении которой возникло исключение, либо на следующую за ней. На этапе знакомства с системами прерываний и исключений такая процедура заметна усложнила бы отладку программы, поэтому в настоящем примере использован не вполне правильный, но более простой способ завершения обработчиков исключений командой перехода на завершение программы. В программе предусмотрено, что в случае возникновения того или иного исключения его номер выводится перед завершением программы на экран. Это дает возможность диагностировать программу, а также исследовать условия возникновения исключений с различными номерами. В следующей статье будет показано, как можно повысить информативность обработчиков исключений.

Затем начинается процедура main. В ней после инициализации регистра DS заполняются дескрипторы сегментов и инициализируется регистр GDTR (предложения 83...109). Эта часть программы полностью повторяет соответствующий фрагмент предыдущего примера.

После запрещения прерываний (предложение 110) можно загрузить регистр IDTR информацией о местонахождении и размере таблицы IDT. Для загрузки IDTR предусмотрена специальная привилегированная команда lidt (load interrupt descriptor table, загрузка таблицы дескрипторов прерываний), которая, как и команда lgdt, требует указания в качестве операнда имени псевдодескриптора. В предложении 111 в псевдодескриптор pdescr заносится граница IDT, далее определяется и заносится в псевдодескриптор линейный базовый адрес IDT, и, наконец, в предложении 116 выполняется загрузка IDTR. Начиная с этого момента процессор будет обрабатывать все прерывания через нашу таблицу IDT. Однако процессор пока работает в реальном режиме, а формат IDT предполагает защищенный режим. Если бы мы не запретили прерывания перед загрузкой регистра IDTR, первое же прерывание (от таймера) привело бы к отключению процессора.

Переход в защищенный режим и инициализация сегментных регистров (предложения 117...125) выполняется, как и в предыдущем примере.

Фрагмент программы, начинающийся с предложения 136, служит диагностическим целям. В регистр AX засылается произвольная константа (в примере – FFFFh), в регистр SI загружается смещение строки string и выполняется вызов подпрограммы wrd_asc, которая преобразует число в AX в символьную форму и помещает результат преобразования (четыре 16-ричные цифры) в строку, адрес которой находится в регистре SI. Все детали такого преобразования подробно обсуждались в статье 13. Таким образом, тройка команд

```
mov     AX,0FFFFh
mov     SI,offset string
call    wrd_asc
```

приводит к помещению в строку по адресу string символьного представления содержимого регистра AX (в данном случае числа FFFF). Указанную комбинацию команд можно использовать в программе неоднократно. Пусть, например, в некоторой точке программы нас интересует состояние указателя стека SP и содержимое верхнего двойного слова стека. Если включить в программу предложения

```
mov     AX,SP         ;Содержимое SP в AX
mov     SI,offset string+5
call    wrd_asc
pop     EAX           ;Двойное слово из стека в EAX
push    EAX           ;И назад, чтобы не смещать стек
mov     SI,offset string+15;Адрес для младшего слова
call    wrd_asc
rol     EAX,16        ;Обменяем половины EAX
mov     SI,offset string+10;Адрес для старшего слова
                      ;(левее младшего слова на экране)
call    wrd_asc
```

то в строку string начиная с байта 5 будет помещено содержимое SP, а начиная с байта 10 – содержимое верхнего двойного слова стека (сначала старшие биты, затем младшие). При необходимости строку string можно удлинить и выводить на экран больше информации.

В предложениях 139...145 строка string пересылается в видеопамять начиная с заданной в регистре DI позиции. Атрибут строки содержится в регистре AH.

Программы преобразования и вывода на экран не требуют системных средств и могут выполняться в защищенном режиме. Таким образом, с помощью подпрограммы wrd_asc можно динамически изучать содержимое регистров, стека и ячеек памяти. Сама подпрограмма (вместе с сопутствующей подпрограммой bin_asc) размещена в самом конце сегмента команд. Текст ее полностью взят из статьи 13.

Первая позиция строки string (4 байта) зарезервирована для вывода при нормальном выполнении программы контрольного числа FFFFh, а при наличии исключения – его номера. При возникновении любого исключения процессор находит в таблице дескрипторов прерываний дескриптор, порядковый номер которого совпадает с номером исключения (дескрипторы прерываний можно рассматривать как векторы прерываний, только не 4-байтовые, как в реальном режиме, а 8-байтовые), извлекает из него селектор и смещение обработчика исключения. Адрес возврата сохраняется в стеке (подробности будут рассмотрены позже), и выполняется переход на программу обработчика (процедуры exc0, exc3 и т. д.). Обработчики исключений нашего примера построены единообразно: в AX засылает-

ся номер исключения и выполняется переход в точку home, где вызывается подпрограмма wrd_asc, после чего строка string выводится на экран.

После вывода диагностической строки осуществляется возврат в реальный режим так же, как и в предыдущем примере. Теневые регистры загружаются значениями, действительными для реального режима (предложения 146...158), далее с помощью регистра CR0 переключается режим и выполняется дальний переход для загрузки регистра CS сегментным адресом сегмента команд.

В реальном режиме помимо восстановления значений DS и SS необходимо выполнить еще одну важную операцию: восстановить содержимое регистра IDTR, действительное для реального режима. Этот регистр служит для определения характеристик таблицы векторов прерываний. При переходе в защищенный режим мы загрузили в него характеристики таблицы дескрипторов прерываний защищенного режима. Теперь в него следует загрузить характеристики таблицы векторов реального режима. Начальный адрес этой таблицы 0, а размер составляет 1 Кбайт (400h байт). Таким образом, граница равна 3FFh. Указанные значения помещаются в псевдодескриптор (предложения 171...174), и командой lidt осуществляется загрузка регистра IDTR. Переход в реальный режим закончен. После разрешения прерываний на экран выводится контрольное сообщение и программа завершается обычным образом.

Статья 68. **Исследование исключений**

Рассмотрим более подробно классификацию прерываний.

Если процессор по каким-либо причинам не может выполнить очередную команду, возникает внутреннее событие, называемое исключением. В зависимости от причины возникновения различают 18 видов исключений, которым присвоены номера векторов от 0 до 17 (строго говоря, исключений только 15 с номерами 0, 1, 3...8, 10...14, 16 и 17; вектор 2 закреплен за немаскируемыми прерываниями, возникающими при поступлении сигнала на ввод NMI микропроцессора, а векторы 9 и 15 не используются). При возникновении исключения процессор умножает его номер на 8 и полученное произведение использует как индекс в таблице дескрипторов прерываний IDT. Таким образом, первые 18 дескрипторов IDT должны всегда описывать программы обработчиков исключений. Следующие 14 векторов с номерами 18...31 зарезервированы для будущих применений.

Различные устройства компьютера (таймер, клавиатура и др.) сигнализируют о необходимости программного вмешательства в их работу с помощью сигнала прерывания, который, пройдя через контроллер прерываний, поступает на вход INT микропроцессора и инициирует в нем выполнение процедуры прерывания. В состав этой процедуры входит, в частности, чтение номера вектора, устанавливаемого контроллером прерываний на шине данных компьютера. Как известно, контроллер прерываний формирует передаваемый в процессор номер вектора путем сложения базового номера, хранящегося в контроллере, с номером линии, по которой поступил запрос от устройства. Номер базового вектора (в принципе в диапазоне 0...255) устанавливается в процессе программной инициализации контроллера. Поскольку в защищенном режиме векторы 0...31 зарезервированы за исключениями, базовые векторы контроллеров прерываний должны быть расположены в диапазоне 32...248. Таким образом, для обра-

ботки внешних аппаратных прерываний в защищенном режиме необходимо перепрограммировать контроллеры прерываний компьютера.

Сигнал немаскируемого аппаратного прерывания, поступающий на вход NMI микропроцессора, возникает в результате серьезного аппаратного сбоя в работе компьютера. Как ужс отмечалось выше, для него зарезервирован вектор прерывания с номером 2.

Программные прерывания возникают при выполнении процессором команды int с числовым аргументом. В принципе этот аргумент может принимать значения от 0 до 255, однако практически для программных прерываний допустимо использовать только векторы, оставшиеся свободными после размещения, во-первых, векторов исключений (32 вектора) и, во-вторых, векторов аппаратных прерываний (16 векторов). При этом два программных прерывания, именно команда int 3, служащая для отладки, и команда into, фиксирующая арифметическое переполнение, обрабатываются в составе таблицы исключений. Для первой выделен вектор с номером 3, для второй – с номером 4.

Нарушения в работе программы, приводящие к исключениям, могут иметь разную природу и разные возможности исправления в процессе выполнения программы. В соответствии с этим исключения подразделяются на три класса: нарушения, ловушки и аварии.

Нарушение, или отказ (fault), – это исключение, фиксируемое еще до выполнения команды или в процессе ее выполнения. Типичными примерами нарушений являются адресация за установленной границей сегмента или обращение к отсутствующему дескриптору. При обработке нарушения процессор сохраняет в стеке адрес той команды, выполнение которой привело к исключению. При этом предполагается, что в обработчике нарушения его причина будет ликвидирована, после чего команда iret вернет управление на ту же, еще не выполненную команду. Таким образом, сам механизм обработки нарушений предполагает восстановление этого программного сбоя.

Ловушка (trap) обрабатывается процессором после выполнения команды, вызвавшей это исключение, и в стеке сохраняется адрес не этой, а следующей команды. Таким образом, после возврата из обработчика ловушки выполняется не команда, инициировавшая исключение, а следующая за ней команда программы. К ловушкам относятся все команды программных прерываний int.

Авария, или выход из процесса (abort), является следствием серьезных невосстановимых ошибок, например обнаружение в системных таблицах неразрешенных или несовместимых значений. Адрес, сохраняемый в стеке, не позволяет локализовать вызвавшую исключение команду, и восстановление программы не предполагается. Обычно аварии требуют перезагрузки системы.

Процессор, зарегистрировав то или иное исключение, сохраняет в стеке содержимое расширенного регистра флагов EFLAGS, селектор сегмента команд, смещение точки возврата, а также (в некоторых случаях) 32-битовый код ошибки (рис. 68.1). Даже если программа выполняется в режиме 16-битовых адресов и операндов, в качестве смещения возврата выступает 32-битовое содержимое указателя команд EIP, в котором старшая половина, очевидно, равна нулю. Стоит упомянуть, что двухсловные данные располагаются в стеке всегда в таком порядке: в слове стека с меньшим адресом – младшая часть 32-битового данного, в слове стека с большим (на 2) адресом – старшая часть. Для выравнивания стека под содержимое CS также отводится двойное слово, в младшей половине которого располагается CS, а старшая половина ничем не заполняется. Для общности на рисунке

обозначен 32-разрядный указатель стека ESP, хотя в режиме 16-разрядных операндов фактически используется только его младшая половина SP.

	Адреса ячеек стека
Код ошибки	m-16
EIP	m-12
"Мусор" CS	m-8
EFLAGS	m-4
	m (исходное состояние ESP)

Рис. 68.1. Состояние стека при исключении

Код ошибки, включаемый для некоторых исключений в стек, позволяет получить дополнительную информацию об исключении, которую может использовать его обработчик. Формат кода ошибки зависит от номера исключения. Для исключений 10, 11, 12 и 13 код ошибки имеет формат, приведенный на рис. 68.2.

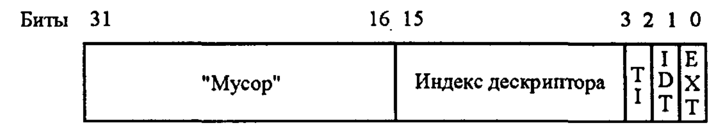


Рис. 68.2. Формат кода ошибки для исключений сегментов

Бит 0 (EXT – от External, внешний) равен единице, если исключение возникло в процессе обработки другого исключения или внешнего прерывания.

Бит 1 (IDT) равен единице, если исключение возникло в процессе чтения элемента таблицы дескрипторов прерываний, что может иметь место только при обработке исключения или прерывания.

Бит 2 (TI – от Table Indicator, индикатор таблицы) равен единице, если дескриптор находится в таблице локальных дескрипторов, и нулю, если дескриптор глобальный. В нашем случае все дескрипторы глобальные, так как таблицы локальных дескрипторов у нас нет.

Перед завершением обработчика (командой iret) код ошибки следует снять со стека.

Таким образом, для правильной обработки исключения необходимо знать причину его возникновения и вид, а также куда вернется управление после завершения обработчика и включен ли в стек код ошибки. В табл. 68.1 приведен список всех исключений с краткими характеристиками.

Таблица 68.1. Исключения процессора

Вектор	*Название исключения*	*Класс исключения*	*Код ошибки*	*Команды, вызывающие исключение*
0	Ошибка деления	Нарушение	Нет	div, idiv
1	Исключение отладки	Нарушение	Нет	Любая команда/ловушка
2	Немаскируемое прерывание	–	–	–
3	int 3	Ловушка	Нет	int 3
4	Переполнение	Ловушка	Нет	into

5	Нарушение границы массива	Нарушение	Нет	bound
6	Недопустимый код команды	Нарушение	Нет	Любая команда
7	Сопроцессор недоступен	Нарушение	Нет	esc, wait
8	Двойное нарушение	Авария	Да	Любая команда
9	Выход сопроцессора из сегмента	Авария	Нет	Команда сопроцессора обращения к памяти
10	Недопустимый сегмент состояния задачи TSS	Нарушение	Да	jmp, call, iret, прерывание
11	Отсутствие сегмента	Нарушение	Да	Команда загрузки сегментного регистра
12	Ошибка обращения к стеку	Нарушение	Да	Команда обращения к стеку
13	Общая защита	Нарушение	Да	Команда обращения к памяти
14	Страничное нарушение	Нарушение	Да	Команда обращения к памяти
15	Зарезервировано	–	–	–
16	Ошибка сопроцессора	Нарушение	Нет	esc, wait
17	Ошибка выравнивания	Нарушение	Да	Команда обращения к памяти
18...31	Зарезервированы			
32...255	Предоставлены пользователю для аппаратных прерываний и команд int			

При переходе на обработчик исключения процессор заносит в стек адрес возврата. В обработчике исключения этот адрес можно извлечь и вывести на экран вместе с номером возникшего исключения. Такое усовершенствование обработчиков существенно упрощает процесс отладки программ защищенного режима, так как позволяет определить, в какой именно точке программы произошло нарушение ее работы. При извлечении из стека адреса возврата следует учитывать возможность наличия в стеке кода ошибки.

Усовершенствованные программы обработчиков приведены ниже:

```
exc0    proc                ;Обработчик исключения 0
        pop   EAX           ;AX=IP
        mov   SI,offset string+5;Выведем адрес
        call  wrd_asc       ;возврата на экран
        mov   AX,0          ;Номер исключения
        jmp   home          ;На выход из обработчика
exc0    endp
```

Так же выглядит и обработчик исключения 3 (разумеется, с отличием в предложении занесения в регистр AX номера исключения). Остальные обработчики (исключений 10…13) будут несколько иными:

```
exc10   proc
        pop     EAX           ;AX=код ошибки (нам не нужен)
        pop     EAX           ;AX=IP
        mov     SI,offset string+5 ;Выведем адрес
        call    wrd_asc       ;возврата на экран
        mov     AX,10         ;Номер исключения
        jmp     home          ;На выход из обработчика
exc10   endp
```

Внеся в программу предыдущей статьи описанные выше усовершенствования, воспользуемся ею для демонстрации причин возникновения исключений. Отлаженная, нормально работающая программа выводит на экран последовательность цветных символов, а также диагностическую строку, которая имеет вид

```
FFFF ****-**** ****-**** ****
```

Если в процессе выполнения программы возникает исключение, на экран вместо числа FFFF выводится 16-ричный номер исключения. Включим в программу различные варианты неправильных программных строк, приводящих к исключениям. Эти строки можно помещать в программу в любом месте, где процессор работает в защищенном режиме, например после предложения 129 или 135. Естественно, перед включением в программу нового исследовательского фрагмента старый надо удалить и, прогнав программу, убедиться в том, что ее работоспособность восстановлена.

При отладке программ защищенного режима чаще всего возникает исключение 13 – нарушения общей защиты. Попробуем обратиться к сегменту данных по относительному адресу, выходящему за его пределы:

```
        mov     AX,DS:[data_size-1];Возникает исключение 13
```

Байт с номером data_size-1 является последним байтом сегмента команд. Если бы мы обратились по этому адресу командой чтения байта

```
        mov     AL,DS:[data_size-1];Команда не вызывает нарушений
```

все было бы нормально. Однако при чтении целого слова второй байт этого слова оказывается за пределами сегмента данных и возникает нарушение общей защиты. Таким образом, в защищенном режиме процессор строго следит за тем, чтобы программа не обращалась к невыделенным ей участкам памяти.

Попробуем теперь прочитать байт из сегмента команд. Атрибут 98h, который мы присвоили сегменту команд, говорит о том, что это исполняемый сегмент, который запрещается читать или модифицировать. Команда

```
        mov     AL,CS[0]      ;Команда вызывает нарушение общей защиты
```

должна прочитать в регистр AL первый байт сегмента команд. Однако команда выполнена не будет, а возбудит нарушение общей защиты. Таким образом, в защищенном режиме коды команд в сегменте команд нельзя ни модифицировать, ни даже читать. Впрочем, это относится только к сегментам с атрибутом 98h. Если же сегменту команд назначить атрибут 9Ah, из него можно будет читать.

Нарушение общей защиты возникает и в том случае, когда программа обращается к памяти за границей таблицы дескрипторов (вспомним, что в регистре таблицы дескрипторов имеется не только поле базы таблицы, но и поле ее границы). Включим в программу строку загрузки в сегментный регистр отсутствующего в таблице глобальных дескрипторов селектора:

```
        mov     AX,40
        mov     FS,AX         ;Команда вызывает нарушение общей защиты
```

Вторая команда этого фрагмента должна загрузить в теневой регистр, связанный с сегментным регистром FS, дескриптор, отвечающий селектору 40, т. е. шестой по счету дескриптор. Однако в нашей таблице только 5 дескрипторов, последний из которых имеет индекс 32, поэтому при выполнении команды возникает исключение общей защиты.

Схожая ситуация возникает, если в программе встречается команда прерывания int с номером, отсутствующим в таблице дескрипторов прерываний. Команда

```
int   20          ;Команда вызывает нарушение общей защиты
```

возбудит исключение общей защиты, поскольку она должна извлечь из таблицы IDT дескриптор с порядковым номером 20, а наша таблица имеет только 18 дескрипторов прерываний. А вот команда

```
int   3           ;Нарушения не будет
```

является совершенно законной, так как для нее в IDT предусмотрен дескриптор, а в сегменте команд соответствующий обработчик (который всего лишь выведет в первую позицию диагностической строки число 0003, а во вторую – смещение команды int 3).

Продемонстрируем теперь исключение нарушения стека. После предложения 129 стек должен находиться в исходном состоянии, т. е. SP=100h. Выведем на экран для контроля содержимое SP и попробуем извлечь из стека одно слово. Поскольку стек пуст, это слово будет извлекаться из области памяти, лежащей за его границей.

```
mov   AX,SP       ;Выведем на экран содержимое SP
mov   SI,offset string+10;для контроля
call  wrd_asc
pop   AX          ;Команда вызывает нарушение стека
```

Возникает нарушение с номером 0Ch – ошибка обращения к стеку.

Рассмотрим еще одно исключение, возникающее при обращении к сегменту, помеченному как отсутствующий (исключение с вектором 11). Это исключение связано с фундаментальным для защищенного режима понятием присутствия (или отсутствия) сегмента. В дескрипторе любого сегмента имеется бит присутствия сегмента в памяти. Для дескрипторов сегментов памяти (т. е. сегментов команд, данных или стека) этот бит, обозначаемый P (от Present, присутствующий), находится в разряде 7 байта атрибутов 1.

Бит присутствия предназначен для организации виртуального режима, в котором суммарный размер всех выполняемых одновременно программ может превышать фактический объем оперативной памяти. В этом случае операционная система хранит часть сегментов программ на диске, загружая их в память по мере необходимости на место тех сегментов, к которым временно не происходит обращений. В частности, система может загружать в память то одну, то другую, то третью задачи, создавая иллюзию их одновременной работы.

Таким образом, работа с битом присутствия – функция операционной системы. Выгрузив какой-то сегмент на диск, система сбрасывает бит P в дескрипторе этого сегмента. Если программа делает попытку обращения к отсутствующему сегменту, возникает исключение, которое в конечном счете должно заставить систему загрузить требуемый сегмент в память, возможно, выгрузив перед этим другой сегмент. После загрузки сегмента в память система устанавливает в его дескрипторе бит P, помечая его присутствующим. Последующие обращения к этому сегменту будут выполняться без нарушений.

Если прикладная программа берет на себя часть функций операционной системы, она может сама устанавливать и сбрасывать бит присутствия, регулируя тем самым

(с помощью исключения отсутствия сегмента) доступ к сегментам. В простом случае, когда все сегменты всегда находятся в памяти, биты присутствия в их дескрипторах должны быть установлены.

Читатель без труда сможет, сбросив бит присутствия в сегменте данных или видеопамяти, убедиться в возникновении исключения 11 в той точке программы, где происходит первое обращение к этому сегменту в защищенном режиме.

Статья 69. **Обработка аппаратных прерываний в защищенном режиме**

В предыдущих статьях мы рассмотрели обработку внутренних прерываний процессора, так называемых исключений. Перейдем теперь к обработке внешних, аппаратных прерываний, которые часто называют просто прерываниями. В принципе обработка прерывания выполняется, за исключением своей начальной аппаратной стадии, так же, как и обработка исключения: при поступлении на вход INT микропроцессора сигнала от внешнего устройства процессор считывает с шины данных выставленный контроллером прерываний номер вектора, находит в таблице дескрипторов прерываний дескриптор с соответствующим номером и, сохранив предварительно в стеке адрес возврата, осуществляет переход на обработчик прерывания. Команда iret, которой заканчивается обработчик прерывания, возвращает управление в программу. Таким образом, для обработки аппаратных прерываний мы должны дополнить таблицу IDT дескрипторами обработчиков аппаратных прерываний и включить сами обработчики в текст программы.

В действительности, однако, дело обстоит несколько сложнее.

Поскольку первые 32 вектора зарезервированы для обработки исключений, аппаратным прерываниям придется назначить другие векторы, например, начиная с номера 32=20h. Однако в машинах типа IBM PC контроллеры прерываний всегда программируются в процессе начальной загрузки так, что базовый вектор ведущего контроллера равен восьми, а ведомого – 70h. Таким образом, перед переходом в защищенный режим нам придется перепрограммировать контроллеры прерываний, назначив им базовые адреса за пределами векторов, предназначенных для обслуживания исключений процессора. В системах Windows ведущему контроллеру назначается базовый вектор 50h, а ведомому – 58h. Мы в наших учебных задачах не связаны правилами Windows и можем назначить ведущему контроллеру, например, вектор 20h, а ведомому – 28h (или оставив у ведомого контроллера базовый вектор 70h). Между прочим, различие базовых векторов в реальном и защищенном режимах является еще одной причиной программной несовместимости этих режимов.

Очевидно, что перед возвратом в реальный режим контроллеры надо снова перепрограммировать, иначе не смогут работать обработчики аппаратных прерываний BIOS. Однако это дело относительно трудоемкое. Можно поступить проще: переход в реальный режим реализовать с помощью отключения процессора, предварительно загрузив в область данных BIOS по адресу 40h:67h двухсловный адрес возврата в нашу программу. Если при этом в байт 0Fh КМОП-микросхемы заслать вместо кода 0Ah, как это мы делали в примере 65.1, код 05h, то после сброса процессора BIOS выполнит перепрограммирование контроллеров прерываний, после чего, как и раньше, передаст управление в указанную нами точку. Поскольку, однако, мы в наших примерах

используем программный переход в реальный режим (путем сброса бита 0 регистра CR0), нам придется перепрограммировать контроллеры прерываний дважды – перед переходом в защищенный режим и перед возвратом в реальный.

Вторая особенность обработки прерываний связана с форматом дескриптора прерываний (шлюза). Как уже отмечалось в статье 67, в таблицу IDT могут входить шлюзы задачи, прерываний и ловушек. Для дескрипторов ловушек мы использовали значение поля атрибута 8Fh; дескрипторы аппаратных прерываний должны иметь атрибут 8Eh.

Так же как и в реальном режиме, если переход на обработчик осуществляется через шлюз прерывания, процессор сбрасывает при входе в обработчик флаг IF в регистре флагов EFLAGS. Команда iret, загружая из стека сохраненное там исходное содержимое EFLAGS, снова разрешает прерывания. При переходе на обработчик через шлюз ловушки состояние флага IF не изменяется.

Модифицируем программу 67.1, введя в нее обработку аппаратных прерываний (пример 69.1). Для простоты ограничимся обработкой прерываний только от двух источников – таймера и клавиатуры. Таймер, будучи постоянно действующим источником прерываний, является естественным объектом исследований, а обработку прерываний от клавиатуры придется включить для того, чтобы нажатия на клавиши например, при вводе команды запуска программы, не приводили к сбоям в защищенном режиме. Таким образом, мы сможем ограничиться перепрограммированием только ведущего контроллера и включением в таблицу IDT лишь двух шлюзов прерываний. Чтобы полностью обезопасить себя от незапланированных прерываний, их можно замаскировать в контроллерах прерываний.

Общий ход выполнения программы выглядит следующим образом. После необходимых инициализирующих действий программа начинает, как и в предыдущих примерах, выводить на экран цепочку символов. Для замедления этого процесса в цикл вывода включена программная задержка, а число шагов цикла увеличено. Прерывания от таймера, пересчитываемые в отношении 4:1, приводят к выводу на экран знака "!". После того как цикл вывода символов в основной программе завершится, программа выводит на экран диагностическую строку, переводит процессор в реальный режим и завершается, выведя перед этим сообщение функцией DOS.

Как уже отмечалось, многие программные блоки будут без изменений переходить из примера в пример. В таких случаях мы будем ограничиваться заголовочными комментариями. Полный текст опущенных блоков можно найти в примере 67.1.

Пример 69.1. Обработка аппаратных прерываний от таймера и клавиатуры

```
.586P                   ;(1)
;Структура descr для описания дескрипторов сегментов
        ...
;Структура trap для описания шлюзов ловушек
        ...
;Сегмент данных
data    segment use16   ;(2)16-разрядный сегмент
;Таблица глобальных дескрипторов GDT
        ...
;Таблица дескрипторов прерываний
idt label word          ;(3)Метка начала таблицы дескрипторов исключений
trap 13 dup (<dummy>)   ;(4)Дескрипторы исключений 0…12
trap <exc13>            ;(5)Дескриптор исключения 13
trap 18 dup (<dummy>)   ;(6)Дескриптор исключений 14…31
trap <new_08,,8Eh>      ;(7)Дескриптор прерывания от таймера
trap <new_09,,8Eh>      ;(8)Дескриптор прерывания от клавиатуры
```

```
idt_size=$-idt              ;(9)Размер таблицы дескрипторов исключений
;Различные данные программы
pdescr  df    0             ;(10)Псевдодескриптор для команд lgdt и lidt
msg db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$';(11)
string  db '**** ****-**** ****-**** ****';(12) Шаблон диагностической строки
;           0    5    10   15   20   25   Позиции в шаблоне
len=$-string                ;(13)Длина строки
mark_08 dw 1600             ;(14)Позиция для вывода из new_08
time_08 db 0                ;(15)Счетчик прерываний
master  db 0                ;(16)Маска прерываний ведущего контроллера
slave   db 0                ;(17)Маска прерываний ведомого контроллера
data_size=$-gdt_null        ;(18)Размер сегмента данных
data    ends                ;(19)
;Сегмент команд
text    segment use16       ;(20)16-разрядный сегмент
        assume CS:text,DS:data;(21)
textseg label word          ;(22)Метка начала сегмента команд
;Обработчик исключения общей защиты
exc13   proc                ;(23)Исключение 13 - общей защиты
        pop    EAX          ;(24)Снимем со стека код ошибки
        pop    EAX          ;(25)EIP в точке исключения
        mov    SI,offset string+5;(26)Место в выводимой строке
        call   wrd_asc      ;(27)Преобразуем в символы
        mov    AX,13        ;(28)Код данного исключения
        jmp    home         ;(29)На завершение программы
exc13   endp                ;(30)
;Обработчик остальных исключений
dummy   proc                ;(31)
        mov    AX,5555h     ;(32)Условный код
        jmp    home         ;(33)
dummy   endp                ;(34)
;Обработчик прерываний от таймера
new_08  proc                ;(35)
        push   AX           ;(36)Сохраним используемые
        push   BX           ;(37)регистры
        test   time_08,03   ;(38)Пересчет на 4, чтобы снизить
        jnz    skip         ;(39)частоту вывода символа на экран
        mov    AL,21h       ;(40)Символ "!"
        mov    AH,71h       ;(41)Цвет
        mov    BX,mark_08   ;(42)Позиция на экране
        mov    ES:[BX],AX   ;(43)Отправим символ в видеопамять
        add    mark_08,2    ;(44)Смещение по экрану
skip:   inc    time_08      ;(45)Пересчет прерываний
        mov    AL,20h       ;(46)EOI ведомого
        out    20h,AL       ;(47)контроллера
        pop    BX           ;(48)Восстановим используемые
        pop    AX           ;(49)регистры
        db     66h          ;(50)Возврат
        iret                ;(51)в программу
new_08  endp                ;(52)
;Обработчик прерываний от клавиатуры
new_09  proc                ;(53)
        push   AX           ;(54)Сохраним AX
        in     AL,60h       ;(55)Получим введенный символ
        in     AL,61h       ;(56)Получим содержимое порта B
        or     AL,80h       ;(57)Установкой старшего бита
        out    61h,AL       ;(58)и последующим сбросом его
        and    AL,7Fh       ;(59)сообщим контроллеру о
        out    61h,AL       ;(60)приеме скан-кода символа
        mov    AL,20h       ;(61)Команда EOI конца
        out    20h,AL       ;(62)прерывания
```

```
        pop    AX            ;(63)Восстановим AX
        db     66h           ;(64)Возврат
        iret                 ;(65)в программу
new_09  endp                 ;(66)
main    proc                 ;(67)
        xor    EAX,EAX       ;(68)
        mov    AX,data       ;(69)
        mov    DS,AX         ;(70)
;Вычислим и загрузим в GDT линейный адрес сегмента данных
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента команд
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента стека
        ...
;Подготовим псевдодескриптор pdescr и загрузим регистр GDTR
        ...
;Сохраним маски прерываний контроллеров
        in     AL,21h        ;(71)
        mov    master,AL     ;(72)Ведущего
        in     AL,0A1h       ;(73)
        mov    slave,AL      ;(74)Ведомого
;Инициализируем ведущий контроллер (базовый вектор теперь 32)
        mov    AL,11h        ;(75)СКИ1: будет СКИ3
        out    20h,AL        ;(76)
        mov    AL,32         ;(77)СКИ2: базовый вектор
        out    21h,AL        ;(78)
        mov    AL,4          ;(79)СКИ3: ведомый подключен к уровню 2
        out    21h,AL        ;(80)
        mov    AL,1          ;(81)СКИ4: 80x86, требуется EOI
        out    21h,AL        ;(82)
        mov    AL,0FCh       ;(83)Маска прерываний
        out    21h,AL        ;(84)
;Запретим все прерывания в ведомом контроллере
        mov    AL,0FFh       ;(85)Маска прерываний
        out    0A1h,AL       ;(86)В порт
;Загрузим IDTR
        mov    word ptr pdescr,idt_size-1 ;(87)Граница
        xor    EAX,EAX       ;(88)
        mov    AX,offset idt;(89)
        add    EAX,EBP       ;(90)Линейный адрес IDT
        mov    dword ptr pdescr+2,EAX;(91)В псевдодескриптор
        lidt   pdescr        ;(92)Загрузка IDTR
;Переходим в защищенный режим
        mov    EAX,CR0       ;(93)Получим содержимое регистра CR0
        or     EAX,1         ;(94)Установим бит защищенного режима
        mov    CR0,EAX       ;(95)Запишем назад в CR0
;--------------------------------------------------;
; Теперь процессор работает в защищенном режиме ;
;--------------------------------------------------;
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:                    ;(96)
;Делаем адресуемыми данные
        ...
;Делаем адресуемым стек
        ...
;Инициализируем ES
        ...
;Выводим на экран тестовую строку символов
        sti                  ;(97)Разрешаем аппаратные прерывания
        mov    DI,1920       ;(98)Начальная позиция на экране
```

```
        mov    CX,320        ;(99)4 строки символов
        mov    AX,1E01h      ;(100)Символ+атрибут
scrn:   stosw                ;(101)Содержимое AX на экран
        inc    AL            ;(102)Инкремент символа
        push   CX            ;(103)Сохраним CX внешнего цикла
        mov    ECX,200000    ;(104)Введем небольшую задержку
delay:  db     67h           ;(105)Чтобы использовать ECX
        loop   delay         ;(106)Цикл задержки
        pop    CX            ;(107)Восстановим CX
        loop   scrn          ;(108)Цикл вывода символов
;Выведем в диагностическую строку код нормального завершения
        mov    AX,0FFFFh     ;(109)Условный код нормального завершения
home:   mov    SI,offset string;(110)Переход из обработчиков исключений
        call   wrd_asc       ;(111)Преобразование AX в символьную строку
;Выводим на экран диагностическую строку
        mov    SI,offset string;(112)
        mov    CX,len        ;(113)
        mov    AH,74h        ;(114)
        mov    DI,1280       ;(115)
scrn1:  lodsb                ;(116)
        stosw                ;(117)
        loop   scrn1         ;(118)
;Вернемся в реальный режим
        cli                  ;(119)Запретим прерывания
;Сформируем и загрузим дескрипторы для реального режима
        ...
;Выполним дальний переход для того, чтобы заново загрузить селектор
;в регистр CS и модифицировать его теневой регистр
        ...
;Переключим режим процессора
go:     ...                  ;(120)
;-----------------------------------------------------;
; Теперь процессор снова работает в реальном режиме ;
;-----------------------------------------------------;
return:                      ;(121)
;Восстановим операционную среду реального режима
        ...
;Восстановим состояние регистра IDTR реального режима
        ...
;Реинициализируем ведущий контроллер прерываний,
;установив в нем базовый вектор 8
        ...
;Восстановим исходные маски прерываний обоих контроллеров
        mov    AL,master     ;(122)Маска прерываний
        out    21h,AL        ;(123)
        mov    AL,slave      ;(124)Маска прерываний
        out    0A1h,AL       ;(125)
        sti                  ;(126)Разрешим аппаратные прерывания
;Работаем в DOS
        mov    AH,09h        ;(127)Проверим выполнение функций DOS
        mov    DX,offset msg;(128)после возврата в реальный режим
        int    21h           ;(129)
        mov    AX,4C00h      ;(130)Завершим программу обычным образом
        int    21h           ;(131)
main    endp                 ;(132)
;Подпрограммы преобразования числа в символьную форму
wrd_asc proc                 ;(133)
        ...
bin_asc proc                 ;(134)
        ...
code_size=$-textseg          ;(135)Размер сегмента команд
```

```
text    ends                ;(136)Конец сегмента команд
;Сегмент стека
        ...
        end main            ;(137)Конец программы
```

Для сокращения размера программы упрощена диагностика исключений. Обычно при отладке программы возникает лишь исключение 13 – общей защиты. Поэтому мы оставили в программе обработчик только этого исключения; как видно из текста программы (предложения 23...30), он выводит на экран номер исключения, а также содержимое указателя команд IP. Для всех остальных исключений предусмотрен общий обработчик, который выводит на экран условный код 5555 (предложения 31...34). Это дает возможность контролировать возникновение всех возможных исключений. Если в процессе отладки программы выяснится, что при ее выполнении возникает исключение с номером, отличным от 13, придется дополнить программу индивидуальными обработчиками каждого исключения, что позволит установить номер и место возникновения исключения.

Дескрипторы в таблице прерываний должны быть расположены по порядку их векторов. Поэтому начинается таблица с 13 одинаковых дескрипторов исключений, имеющих в нашей программе общий обработчик dummy. Затем идет дескриптор исключения 13, а за ним еще 18 одинаковых дескрипторов. Наконец, на 33-м месте (вектор 32) описан дескриптор обработчика аппаратного прерывания от таймера (new_08), а на следующем месте – дескриптор обработчика аппаратного прерывания от клавиатуры. Как уже отмечалось выше, дескрипторы аппаратных прерываний отличаются от дескрипторов исключений только типом шлюза. Это дало нам возможность определить в таблице IDT оба дескриптора аппаратных прерываний с помощью той же структуры trap, задав для 4-го параметра значение 8Eh (вместо 8Fh, указанного в определении структуры).

В сегмент данных дополнительно включены переменные для обслуживания обработчика от таймера (mark_08 и time_08), а также для хранения исходных значений масок прерываний (master и slave) ведущего и ведомого контроллеров прерываний.

В сегменте команд обработчики прерываний и исключений, так же как и все остальные процедуры, могут следовать в любом порядке. Наш сегмент команд начинается с обработчиков исключений и аппаратных прерываний. Далее расположена главная процедура main, а за ней подпрограммы wrd_asc и bin_asc, необходимые для преобразования двоичного содержимого регистров в символьную форму.

Обработчик прерываний от таймера (предложения 35...52), как уже говорилось, при каждом своем вызове выводит на экран знак ! и смещает позицию, подготавливая ее для следующего вызова. Процедура обработчика начинается с сохранения используемых в ней регистров, после чего выполняется проверка содержимого ячейки time_08 (предложение 38). Программа обработчика продолжается, только если в этой ячейке сброшены оба младших бита. Тем самым осуществляется пересчет прерываний в отношении 4:1. На экран выводится цветной символ !, и выполняются инкременты позиции на экране (в ячейке mark_08) и счетчика прерываний time_08. Обработчик завершается генерацией сигнала конца прерывания EOI для ведущего контроллера (порт 20h), восстановлением сохраненных ранее регистров и командой iret. Поскольку в защищенном режиме аппаратное прерывание смещает стек не на три слова, как в реальном режиме, а на 6 (см. рис. 68.1), обработчики прерываний должны заканчиваться "длинной" командой iret, которая снимет со стека три двойных слова. Для того чтобы в 16-разрядном приложении создать такую

команду, следует перед обычной командой iret использовать префикс замены размера операнда 66h (предложения 50 и 51).

Обработчик прерываний от клавиатуры (предложения 53...66) в нашем примере не выполняет никакой полезной работы, однако необходим для правильного функционирования программы. В нем выполняется стандартный набор операций, состав которых определяется особенностями контроллера клавиатуры.

В предложении 55 выполняется чтение из порта 60h поступившего с клавиатуры кода символа (практически в порту может находиться скан-код отпускания клавиши Enter, оставшийся там от последнего нажатия этой клавиши при запуске нашей программы с клавиатуры). Далее предложениями 56...58 выполняется установка в порту 61h управления клавиатурой старшего бита, а следующими двумя предложениями – сброс этого бита. Кратковременная установка бита 7 в порту 61h оповещает контроллер о том, что программа извлекла код символа из порта 60h; это разрешает контроллеру вывод в порт очередного символа. После посылки в контроллер прерываний команды EOI и восстановления сохраненного ранее регистра AX программа обработчика завершается "длинной" командой iret.

С предложения 67 начинается главная процедура main. В ней обычным образом заполняются глобальные дескрипторы, загружается регистр GDTR, запрещаются внешние прерывания. В предложениях 71...74 сохраняются исходные для данного компьютера значения масок прерываний обоих контроллеров прерываний.

Далее выполняется перепрограммирование (инициализация) ведущего контроллера прерываний с целью изменения его базового вектора. Процедура инициализации рассматривалась в статье 26; здесь она для наглядности приведена еще раз. Закончив инициализацию контроллера, необходимо установить в нем требуемую маску прерываний. Таймер подключен к уровню 0, а клавиатура – к уровню 1, поэтому слово маски, посылаемое в порт 20h, равно FCh.

Инициализация ведомого контроллера, от которого прерывания поступать заведомо не будут, в программе не выполняется, но для надежности все прерывания в нем маскируются (предложения 85 и 86). Теперь, когда вся система прерываний настроена, можно загрузить регистр IDTR, "подложив" процессору новую таблицу дескрипторов прерываний и исключений. Наконец, установкой бита 0 управляющего регистра CR0 процессор переводится в защищенный режим.

Программа защищенного режима принципиально не отличается от примера 67.1, за исключением того, что командой sti разрешаются аппаратные прерывания (предложение 97). В цикл вывода на экран последовательности символов включена небольшая задержка (предложения 103...107). Кроме этого, число выводимых символов увеличено до 320 (предложение 99). Все это нужно для того, чтобы за время выполнения цикла вывода символов таймер, который работает с частотой 18,2 Гц, успел дать несколько сигналов прерываний.

Если программа подготовлена без ошибок, она будет функционировать следующим образом. В 13-ю и последующие строки экрана выводятся с небольшой скоростью желтые символы по синему фону. Их количество (320, т. е. 4 строки) задано в программе. Одновременно в 11-ю строку с частотой 18,2/4 Гц поступают изображения восклицательного знака (синие по белому фону). Перед завершением программы в 9-ю строку экрана выводится диагностическая строка

```
FFFF ****-**** ****-**** ****
```

и, наконец, после перехода процессора в реальный режим в позицию курсора красным по зеленому выводится строка

```
Вернулись в реальный режим!
```

Вид экрана после выполнения рассмотренного примера (который был запущен из оболочки DOS Norton Commander) показан на рис. 69.1.

```
                                                                          5 40р




FFFF ****-**** ****-**** ****

!!!!!!!!

☺☻♥♦♣♠•◘○◙♂♀♪♫☼►◄↕‼¶§▬↨↑↓→←∟↔▲▼ !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOP
QRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~⌂АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯа
бвгдежзийклмноп░▒▓│┤╡╢╖╕╣║╗╝╜╛┐└┴┬├─┼╞╟╚╔╩╦╠═╬╧╨╤╥╙╘╒╓╫╪┘┌█▄▌▐▀рстуфхцчшщъыьэюяЁ
ёЄєЇїЎў°∙·√№¤■  ☺☻♥♦♣♠•◘○◙♂♀♪♫☼►◄↕‼¶§▬↨↑↓→←∟↔▲▼ !"#$%&'()*+,-./0123456789:;<=>?@




F:\CURRENT>p.exe
  Вернулись в реальный режим!
F:\CURRENT>
1Помощь 2Вызов  3Чтение 4Правка 5Копия  6НовИмя 7НовКат 8Удал-е 9Меню   10Выход
```

Рис. 69.1. Вывод программы 69.1

После завершения цикла вывода символов (предложения 112...118) необходимо вернуть систему в первоначальное состояние. Начать эту процедуру следует с запрещения аппаратных прерываний (предложение 119), после чего выполнить рассмотренные ранее операции изменения информации в теневых регистрах дескрипторов, переключения процессора в реальный режим, восстановления операционной среды реального режима (содержимого всех сегментных регистров, а также указателя стека) и, наконец, восстановления исходной таблицы векторов прерываний реального режима. Далее следует заново инициализировать ведущий контроллер прерываний, чтобы установить в нем в качестве базового вектор 8. Фрагмент инициализации в примере 69.1 опущен, так как он полностью повторяет предложения 75...82 за исключением предложения 77, в котором следует указать вектор 8.

После перепрограммирования ведущего контроллера можно установить исходные маски в обоих контроллерах (предложения 122...125) и, наконец, разрешить аппаратные прерывания (предложение 126). Возврат в реальный режим завершен.

Рассмотренный пример позволяет поставить поучительный эксперимент, демонстрирующий возникновение исключений. Если в предложении 99 увеличить число выводимых символов с 320 до 1040, то программа заполнит ими весь экран до последнего знакоместа, но в остальном она будет функционировать как и прежде. Если же задать вывод 1041 символа, последний символ должен быть выведен уже за пределами видеостраницы. Видеопамять там есть (видеопамять всех восьми страниц является

слитным образованием), однако мы задали в дескрипторе размер сегмента точно 4 000 байт и попытка вывода символа за пределами сегмента должна привести к нарушению общей защиты. Так оно и будет; на экран будет выведена диагностическая строка

000D 0000-0107 ****-**** ****

Здесь 000D – номер возникшего исключения (исключение 13 общей защиты), а 00000107 – смещение команды, возбудившей это исключение. По листингу трансляции нетрудно установить, что это команда

```
scrn:   stosw               ;(101)Содержимое AX на экран
```

в которой сделана попытка обратиться к видеопамяти, находящейся за пределами объявленного нами сегмента.

Статья 70. **Переключение задач**

Важнейшей особенностью защищенного режима является возможность параллельного выполнения нескольких задач с надежной защитой их друг от друга, чем и объясняется название режима. Под задачей (task) понимают выполняемую на компьютере программу или ее фрагмент, имеющий относительно самостоятельное значение. В частности, задачей можно назвать всю программу, и тогда многозадачность обозначает параллельное выполнение нескольких не связанных друг с другом программ. Однако задачей может быть и достаточно самостоятельная часть программы. Например, обработчики аппаратных прерываний уместно назвать задачами: для них характерно выполнение параллельно и независимо от основной части программы. Вообще, ничто не мешает включить в состав программы несколько самостоятельных участков, переключение между которыми будет осуществляться периодически или по каким-то событиям, например нажатиям определенных клавиш. Эти фрагменты могут образовывать отдельные сегменты команд, но могут входить в состав единого программного сегмента, как в последующих примерах. В настоящей статье будут рассмотрены детали архитектуры процессора, обеспечивающие выполнение и переключение задач.

Организация многозадачного режима опирается на следующие аппаратные и программные средства, о которых пока не было речи:

- сегмент состояния задачи (Task State Segment, TSS);
- дескриптор сегмента состояния задачи;
- регистр задачи (Task Register, TR);
- дескриптор шлюза задачи (Task gate).

Сегмент состояния задачи (TSS) представляет собой поле данных, включаемое в состав сегмента данных либо образующее отдельный сегмент небольшого размера. Каждая задача, участвующая в процедуре переключения, должна иметь свой TSS; именно наличие TSS делает данный программный объект задачей. В предыдущих примерах сегменты состояния задач отсутствовали и эти программы, таким образом, задачами не являлись. Сегменты состояния задач служат для хранения, при переключениях задач, их текущих контекстов, т. е. содержимого аппаратных регистров процессора и другой информации. Поскольку TSS представляется процессору отдельным сегментом, ему должен соответствовать дескриптор. В отличие от обычных сегментов

данных, TSS описывается не дескриптором сегмента памяти, а системным дескриптором, который к тому же может находиться только в таблице глобальных дескрипторов. Системные дескрипторы имеют практически тот же формат, что и дескрипторы памяти (см. рис. 66.1), отличаясь только отсутствием бита умолчания D и кодом типа дескриптора (0 вместо 1, рис. 70.1).

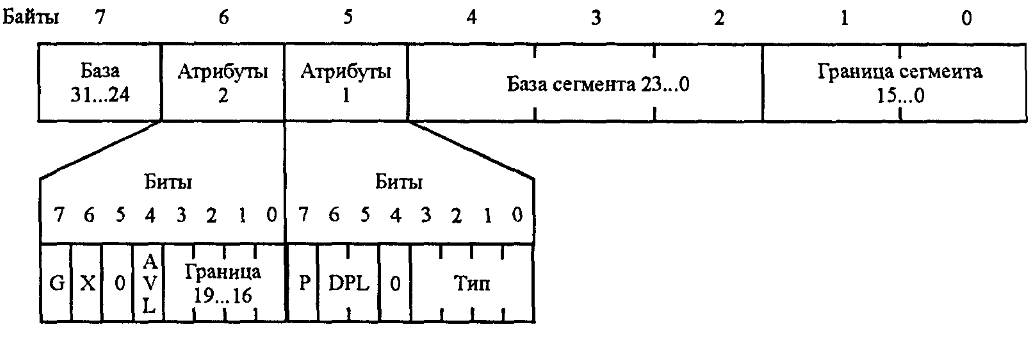


Рис. 70.1. Формат системного дескриптора

Тип системного дескриптора может принимать ряд значений, приведенных в табл. 70.1.

Таблица 70.1. Значения поля типа для системных дескрипторов

Тип	*Назначение дескриптора*
0	Не определено
1	Свободный сегмент состояния задачи (TSS) 80286
2	LDT
3	Занятый сегмент состояния задачи (TSS) 80286
8	Не определено
9	Свободный сегмент состояния задачи (TSS)
Ah	Не определено
Bh	Занятый сегмент состояния задачи (TSS) 80386, i486, Pentium
Dh	Не определено

Любопытно отметить, что возможные для системных дескрипторов значения поля типа не перекрываются значениями для шлюзов (см. табл. 67.1), а дополняют их, хотя форматы тех и других дескрипторов совершенно разные.

В зависимости от порядкового номера дескриптора TSS в таблице дескрипторов ему соответствует тот или иной селектор. Селектор TSS текущей (активной) задачи должен быть загружен в регистр задачи TR. Для исходной, главной задачи эта загрузка осуществляется программно с помощью предназначенной для этого команды ltr (load task register, загрузка регистра задачи); при переключении на новую задачу программа передает процессору селектор нового TSS и перезагрузку регистра TR осуществляет процессор в ходе переключения задач.

Переключение на новую задачу осуществляется командой дальнего вызова call dword ptr или, в некоторых случаях, дальнего перехода jmp dword ptr. В качестве аргумента этих команд указывается двухсловное поле, в первом слове которого записывается селектор тре-

буемого TSS. Существует и другой способ переключения – не через селектор TSS, а через шлюз задачи (дескриптор шлюза с кодом типа, равным пяти). В этом случае селектор требуемого TSS указывается в поле для селектора в шлюзе задачи. Переключение через шлюз задачи имеет то преимущество, что его можно выполнить по аппаратному прерыванию, так как, в отличие от дескриптора сегмента состояния задачи TSS, шлюз задачи можно расположить в таблице дескрипторов прерываний.

Процесс переключения на новую задачу изображен на рис. 70.2.

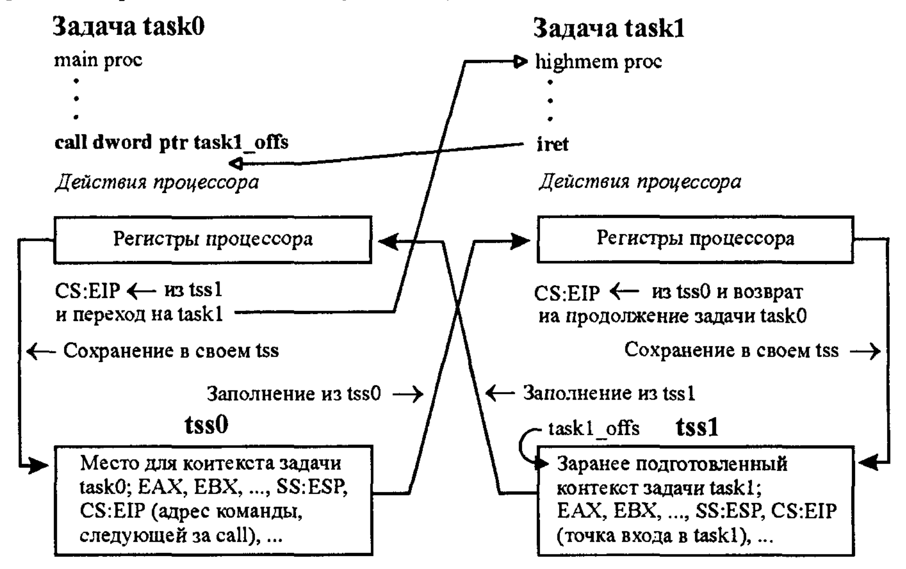


Рис. 70.2. Основные действия при переключении задач

Выполняя переключение задач, процессор сохраняет контекст текущей задачи (task0 на рис. 70.1) в ее TSS и загружает контекст новой задачи, включая селектор и относительный адрес точки входа в задачу, из TSS новой задачи (task1 на рис. 70.2).

Задача, на которую осуществляется переключение, должна в этом случае завершаться командой iret, которая обрабатывается процессором не так, как iret обычного обработчика прерываний. В случае переключения задач процессор по команде iret выполняет обратное перемещение контекстов – контекст завершающейся задачи на момент ее завершения сохраняется в ее TSS, а в регистры процессора из TSS исходной задачи загружается сохраненный там контекст, соответствующий моменту переключения на вторую задачу. Таким образом, TSS можно рассматривать как функциональный аналог стека, который тоже используется для сохранения содержимого регистров и другой информации в обработчиках прерываний и подпрограммах. Однако сохранение в стеке и восстановление из него выполняются с помощью последовательностей команд процессора, а сохранение и восстановление контекстов с помощью TSS осуществляется процессором аппаратно в процессе переключения задач.

Поля TSS исходной задачи заполняются процессором при первом переключении на новую задачу (чтобы можно было вернуться в исходную); программист может не заботиться о его содержимом. Другое дело TSS задачи, на которую осуществляется переключение. В TSS содержится такая важная для выполнения задачи информация, как адрес точки входа, а также исходное содержимое сегментных регистров и регистров общего назначения. По крайней мере некоторые из этих полей должны быть заполнены в исходной задаче еще перед переключением на новую. Рассмотрим кратко содержимое TSS (рис. 70.3).

TSS имеет размер минимум 104 байта. В начале TSS имеется 16-битовое поле связи, используемое при переключении задач. Для исходной, главной задачи (task0) его содержимое не имеет значения. При переключении на новую задачу (task1) процессор заносит в поле связи TSS новой задачи селектор TSS исходной задачи, чем создается связь между новой и старой задачами. Если новая задача, в свою очередь, переключается на следующую задачу (task2), в поле связи TSS этой следующей задачи процессор заносит селектор TSS предыдущей задачи и т. д. В результате создается связный список вложенных задач (рис. 70.4).

			Смещение от базы TSS
0	Связь		00h
ESP0			04h
0	SS0		08h
ESP1			0Ch
0	SS1		10h
ESP2			14h
0	SS2		18h
Регистр управления CR3			1Ch
Указатель команд EIP			20h
Регистр флагов EFLAGS			24h
EAX			28h
ECX			2Ch
EDX			30h
EBX			34h
ESP			38h
EBP			3Ch
ESI			40h
EDI			44h
0	ES		48h
0	CS		4Ch
0	SS		50h
0	DS		54h
0	FS		58h
0	GS		5Ch
0	LDT		60h
Адрес карты ввода-вывода	00000.....0000	T	64h
Карта ввода-вывода (если она есть)			68h

Рис. 70.3. Формат сегмента состояния задачи (TSS)

Команда iret, которой завершается каждая вложенная задача, выполняет обратное переключение задач. В ходе этой операции процессор извлекает из поля связи TSS текущей задачи селектор TSS предыдущей (по порядку вложенности) и передает ей управление, восстановив перед этим из TSS этой задачи ее контекст.

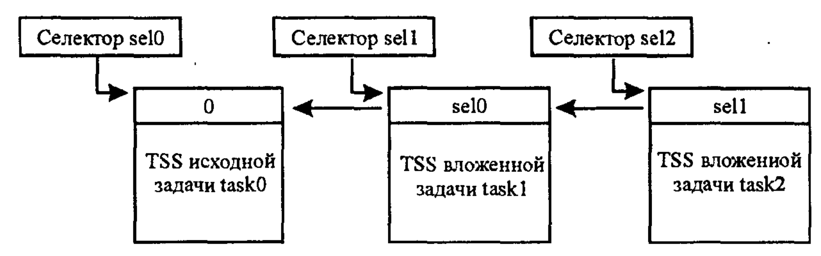


Рис. 70.4. Цепочка связанных TSS

По адресам 04h, 0Ch и 14h относительно базы TSS располагаются кадры стеков уровней привилегий 0, 1 и 2. Содержимое этих полей загружается в регистры SS и ESP, если при переключении задачи происходит смена уровня привилегий. Задачи наших примеров будут работать на одном (нулевом) уровне привилегий; в этом случае поля кадров стеков не используются, а регистры SS и ESP инициализируются содержимым ячеек TSS с адресами 50h и 38h.

Регистр управления CR3 (поле TSS с адресом 1Ch) содержит базовый адрес каталога страниц и используется, если включено страничное преобразование. Наличие в TSS поля для регистра CR3 позволяет иметь для каждой задачи свой каталог страниц и, соответственно, свои таблицы отображения виртуальных адресов, действующих в программе, на физические адреса памяти. В наших примерах страничное преобразование выключено и регистр CR3 не используется.

Двухсловное поле TSS с адресом 20h предназначено для хранения значения указателя команд EIP. В TSS исходной задачи это поле заполняется процессором при переключении на новую задачу; процессор заносит туда адрес команды, следующей за командой переключения, т. е. адрес возврата. В TSS задачи, на которую происходит переключение, поле для EIP должно быть заполнено программно смещением точки входа в задачу. При этом следует иметь в виду, что при возврате в старую задачу командой iret процессор записывает в поле для EIP завершившейся задачи в качестве "адреса возврата" адрес команды, следующей за iret, т. е. адрес уже за пределами задачи. Если планируется повторное переключение на эту задачу, то перед каждым переключением необходимо восстанавливать в ее TSS адрес точки входа.

Сохранение в TSS исходной задачи текущего содержимого регистра флагов EFLAGS (по адресу 24h) позволяет осуществлять переключение в любой точке задачи без потери ее работоспособности.

Участок TSS с адресами 28h...47h отведен для хранения содержимого регистров общего назначения. При переключении с исходной задачи на новую задачу процессор сохраняет в этих полях TSS исходной задачи текущее содержимое регистров, а при обратном переключении командой iret восстанавливает регистры из этих полей TSS исходной задачи, обеспечивая правильное продолжение ее выполнения. Что же касается TSS новой задачи, то, заполнив заранее поля регистров в TSS новой задачи, можно передать ей исходные параметры.

Младшие половины шести двухсловных полей начиная с адреса 48h отведены под содержимое сегментных регистров. Эти поля используются точно так же, как и поля регистров общего назначения.

Если новая задача использует таблицу локальных дескрипторов, ее селектор следует занести в TSS по адресу 60h.

Бит 0 слова по адресу 64h используется для отладки переключаемых задач. Если в TSS новой задачи установлен этот бит, то сразу же после переключения генерируется исключение отладки с номером 1. Остальные биты слова по адресу 64h должны быть равны нулю.

Слово с адресом 66h содержит смещение битовой карты ввода-вывода, которая, при ее наличии, располагается в TSS по последующим адресам и используется для защиты портов компьютера от несанкционированного доступа. Каждый бит этой карты соответствует одному порту (вся карта, таким образом, может достигать 64 Кбит, или 8 Кбайт). Если бит, закрепленный за некоторым портом, равен нулю, задача любого уровня привилегий может обращаться к этому порту. Если бит равен единице, то при обращении к порту задачей с недостаточно высоким уровнем привилегий генерируется исключение общей защиты.

Как уже отмечалось выше, переключение на новую задачу осуществляется с помощью команды дальнего вызова, а возврат в исходную задачу – командой iret. При этом команда iret должна инициировать довольно сложную процедуру обратного переключения через селектор TSS, хранящийся в поле связи TSS текущей задачи. Однако в обработчиках прерываний и исключений та же команда iret выполняется иначе: она просто снимает со стека три 32-битовых слова (EFLAGS, CS и EIP) и загружает их в соответствующие регистры, обеспечивая возврат из обработчика в прерванную задачу. Каким же образом команда iret определяет, в каком режиме ей надлежит работать?

Режим выполнения команды iret определяется состоянием специального флага NT (Nested Task, вложенная задача), расположенного в бите 14 регистра флагов EFLAGS. Команда iret анализирует состояние флага NT и, если он сброшен, осуществляет обычный возврат из программы обработки прерывания (через стек); если же флаг NT установлен, команда iret инициирует обратное переключение задач через селектор в TSS.

После загрузки компьютера флаг NT находится в установленном состоянии. Однако любое аппаратное прерывание или исключение сбрасывает этот флаг, в результате чего команда iret, завершающая обработчик, выполняется в "облегченном" варианте. То же происходит при выполнении процессором команды программного прерывания int. Поскольку команда iret восстанавливает исходное состояние регистра флагов, после завершения обработчика флаг NT снова оказывается установленным.

При выполнении процедуры переключения на новую задачу через шлюз задачи или непосредственно через TSS процессор сохраняет в TSS текущей задачи слово флагов и устанавливает в регистре флагов бит NT (независимо от того, был ли он перед этим сброшен или установлен). Команда iret, завершающая задачу, обнаруживает NT=1 и вместо осуществления возврата через стек инициирует механизм обратного переключения задач.

Если вложенная задача, в свою очередь, выполняет переключение на следующую задачу, текущее слово флагов с установленным битом NT сохраняется в TSS текущей задачи. После завершения новой задачи это слово будет возвращено в регистр флагов и, таким образом, задача будет продолжаться с NT=1, что обеспечит ее правильное завершение.

Рассмотрим пример, демонстрирующий механизм переключения задач (пример 70.1). Для этого включим в состав нашей программы две практически одинаковые процедуры – подпрограммы task1 и task2, выводящие на экран некоторые контрольные строки. Главная процедура, main будет представлять собой исходную управляющую задачу. Процедуры task1 и task2 будут выполнять роль задач, на которые осуществляется переключение из управляющей задачи. В примере 70.1 тексты этих процедур вы-

делены в отдельные сегменты команд (которых, таким образом, будет в программе три). Это сделано только для наглядности; процедуры task1 и task2 можно было расположить и в общем сегменте команд. Главная процедура осуществляет все необходимые инициализирующие действия, переводит процессор в защищенный режим и переключается последовательно сначала на задачу task1, а затем на задачу task2. Далее обычным образом подготавливается среда реального режима и выполняется переключение в реальный режим. Вывод программы в случае ее правильной работы приведен на рис. 70.5.

```
FFFF ****-**** ****-**** ****

Работает процедура Task1      Работает процедура Task2
```

Рис. 70.5. Вывод программы 70.1

Пример 70.1. Задачи и их переключение

```
.586P                        (1)
;Структура для описания дескрипторов сегментов
       ...
;Структура для описания шлюзов ловушек
       ...
;Сегмент данных
data    segment use16     ;(2)16-разрядный сегмент
;Таблица глобальных дескрипторов GDT
gdt_null descr <>          ;(3)Селектор 0
gdt_data descr <data_size-1,,,92h>;(4)Селектор 8, сегмент данных
gdt_code descr <code_size-1,,,98h>;(5)Селектор 16, сегмент команд
gdt_stack descr <599,,,92h>;(6)Селектор 24, сегмент стека
gdt_screen descr <3099,8000h,0Bh,92h>;(7)Селектор 32, видеопамять
gdt_text1 descr <text1_size-1,,,9Ah>;(8)Селектор 40, коды task1
gdt_text2 descr <text2_size-1,,,9Ah>;(9)Селектор 48, коды task2
gdt_tss0 descr <103,0,0,89h>;(10)Селектор 56, TSS0
gdt_tss1 descr <103,0,0,89h>;(11)Селектор 64, TSS1
gdt_tss2 descr <103,0,0,89h>;(12)Селектор 72, TSS2
gdt_size=$-gdt_null        ;(13)Размер GDT
;Таблица дескрипторов прерываний
idt label word             ;(14)Метка начала таблицы дескрипторов исключений
trap 10 dup (<dummy>)      ;(15)Дескрипторы исключения 0…9
trap <exc10>               ;(16)Дескриптор исключения 10
trap <exc11>               ;(17)Дескриптор исключения 11
trap <exc12>               ;(18)Дескриптор исключения 12
trap <exc13>               ;(19)Дескриптор исключения 13
trap 18 dup (<dummy>)      ;(20)Дескрипторы исключений 14…31
idt_size=$-idt             ;(21)Размер таблицы дескрипторов исключений
;Различные данные программы
pdescr  df    0            ;(22)Псевдодескриптор для команд lgdt и lidt
msg db 27,'[31;42m  Вернулись в реальный режим!  ',27,'[0m$';(23)
string  db '**** ****-**** ****-**** ****';(24)
;           0    5    10   15   20   25   Позиции в шаблоне
len=$-string               ;(25)
tss0    dw 52 dup (0)      ;(26)TSS задачи 0
tss1    dw 52 dup (0)      ;(27)TSS задачи 1
tss2    dw 52 dup (0)      ;(28)TSS задачи 2
task1_offs dw 0            ;(29)
task1_sel dw 64            ;(30)
task2_offs dw 0            ;(31)
task2_sel dw 72            ;(32)
```

```
data_size=$-gdt_null       ;(33)Размер сегмента данных
data    ends               ;(34)
;Сегмент команд
text    segment use16      ;(35)16-разрядный сегмент
        assume CS:text,DS:data;(36)
textseg label word         ;(37)Метка начала сегмента команд
exc10   proc               ;(38)Исключение недопустимого TSS
        mov   AX,8         ;(39)Загрузим в DS селектор сегмента
        mov   DS,AX        ;(40)данных главной задачи
        pop   EAX          ;(41)
        pop   EAX          ;(42)
        mov   SI,offset string+5;(43)
        call  wrd_asc      ;(44)
        mov   AX,10        ;(45)
        jmp   home         ;(46)
exc10   endp               ;(47)
exc11   proc               ;(48)Исключение отсутствия сегмента
        ...
exc11   endp               ;(49)
exc12   proc               ;(50)Исключение обращения к стеку
        ...
exc12   endp               ;(51)
exc13   proc               ;(52)Исключение общей защиты
        ...
exc13   endp               ;(53)
dummy   proc               ;(54)
        mov   AX,8         ;(55)
        mov   DS,AX        ;(56)
        mov   AX,5555h     ;(57)Код остальных исключений
        jmp   home         ;(58)
dummy   endp               ;(59)
main    proc               ;(60)
        xor   EAX,EAX      ;(61)
        mov   AX,data      ;(62)
        mov   DS,AX        ;(63)
;Вычислим и загрузим в GDT линейный адрес сегмента данных
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента команд text
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента команд text1
        xor   EAX,EAX      ;(64)Очистим EAX
        mov   AX,text1     ;(65)Сегментный адрес сегмента команд
        shl   EAX,4        ;(66)
        mov   BX,offset gdt_text1;(67)
        mov   [BX].base_l,AX;(68)
        shr   EAX,16       ;(69)
        mov   [BX].base_m,AL;(70)
;Вычислим и загрузим в GDT линейный адрес сегмента команд text2
        ...                ;По аналогии с предыдущим блоком
;Вычислим и загрузим в GDT линейный адрес сегмента стека
        ...
;Вычислим и загрузим в GDT линейный адрес TSS0
        mov   EAX,EBP      ;(71))Линейный адрес сегмента данных
        add   AX,offset tss0;(72)Линейный адрес TSS0
        mov   BX,offset gdt_tss0;(73)Смещение дескриптора
        mov   [BX].base_l,AX;(74)Загрузим младшую половину базы
        shr   EAX,16       ;(75)AX=старшая половина базы
        mov   [BX].base_m,AL;(76)Загрузим среднюю часть базы
;Вычислим и загрузим в GDT линейный адрес TSS1
        ...                ;По аналогии с предыдущим блоком;
;Вычислим и загрузим в GDT линейный адрес TSS2
```

```
        ...                     ;По аналогии с предыдущим блоком
;Подготовим псевдодескриптор pdescr для загрузки регистра GDTR
        ...
        cli                     ;(77)Запрет прерываний
;TSS0 инициализировать не надо. Инициализируем TSS1
        mov    tss1+4Ch,40 ;(78)CS
        mov    tss1+20h,offset task1;(79)IP
        mov    tss1+50h,24 ;(80)SS
        mov    tss1+38h,200;(81)SP
        mov    tss1+54h,40 ;(82)DS(=CS)
        mov    tss1+48h,32 ;(83)ES
;Инициализируем TSS2
        mov    tss2+4Ch,48 ;(84)CS
        mov    tss2+20h,offset task2;(85)IP
        mov    tss2+50h,24 ;(86)SS
        mov    tss2+38h,400;(87)SP
        mov    tss2+54h,48 ;(88)DS(=CS)
        mov    tss2+48h,32 ;(89)ES
;Загрузим IDTR
        ...
;Переходим в защищенный режим
        mov    EAX,CR0      ;(90)Получим содержимое регистра CR0
        or     EAX,1        ;(91)Установим бит защищенного режима
        mov    CR0,EAX      ;(92)Запишем назад в CR0
;----------------------------------------------;
; Теперь процессор работает в защищенном режиме ;
;----------------------------------------------;
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:                   ;(93)
;Делаем адресуемыми данные и стек; инициализируем ES
        ...
;Загрузим регистр задачи TR селектором TSS главной задачи
        mov    AX,56        ;(94)
        ltr    AX           ;(95)
;Выполним переключение задач
        call   dword ptr task1_offs ;(96)На задачу 1
        call   dword ptr task2_offs ;(97)На задачу 2
;Выведем в диагностическую строку код нормального завершения
        mov    AX,0FFFFh    ;(98)
home:   mov    SI,offset string ;(99)
        call   wrd_asc      ;(100)
;Выведем на экран диагностическую строку
        ...
;Вернемся в реальный режим
;Сформируем и загрузим дескрипторы для реального режима
        ...
;Выполним дальний переход для того, чтобы заново загрузить селектор
;в регистр CS и модифицировать его теневой регистр
        ...
;Переключим режим процессора
go:     ...                 ;(101)
;---------------------------------------------------;
; Теперь процессор снова работает в реальном режиме ;
;---------------------------------------------------;
return:                     ;(102)
;Восстановим вычислительную среду реального режима
        mov    AX,data      ;(103)
        mov    DS,AX        ;(104)
        mov    AX,stk       ;(105)
        mov    SS,AX        ;(106)
```

```
        mov    SP,600         ;(107)
;Восстановим состояние регистра IDTR реального режима
;и завершим программу
        mov    AX,0FFFFh    ;(108)Граница сегмента реального режима
        mov    word ptr pdescr,AX ;(109)
        mov    EAX,0         ;(110)Смещение таблицы векторов
        mov    dword ptr pdescr+2,EAX;(111)
        lidt   pdescr        ;(112)Загрузим псевдодескриптор в IDTR
        sti                  ;(113)Разрешим аппаратные прерывания
        mov    AH,09h        ;(114)Проверим выполнение функций DOS
        mov    DX,offset msg;(115)после возврата в реальный режим
        int    21h           ;(116)
        mov    AX,4C00h      ;(117)Завершим программу обычным образом
        int    21h           ;(118)
main    endp                 ;(119)
;Подпрограммы wrd_asc и bin_asc преобразования числа в символьную форму
        ...
code_size=$-textseg          ;(120)Размер сегмента команд text
text    ends                 ;(121)
;Сегмент стека главной задачи
stk     segment use16 stack;(122)16-разрядный сегмент
        db 600 dup ('^')     ;(123)Увеличенный размер стека
stk     ends                 ;(124)
;Сегмент команд задачи 1
text1   segment use16        ;(125)
        assume CS:text1      ;(126)
task1   proc                 ;(127)Процедура задачи 1
        mov    AH,1Eh        ;(128)Атрибут символов на экране
        mov    SI,offset msg1;(129)Смещение выводимой строки
        mov    DI,1600       ;(130)Позиция на экране
        mov    CX,24         ;(131)Длина выводимой строки
c1:     lodsb                ;(132)Символ строки в AL
        stosw                ;(133)Символ+атрибут на экран
        loop   c1            ;(134)Цикл по строке
        iret                 ;(135)Завершение задачи 1
msg1    db 'Работает процедура Task1';(136)Сообщение задачи 1
task1   endp                 ;(137)Конец процедуры задачи 1
text1_size=$-task1           ;(138)Размер сегмента задачи 1
text1   ends                 ;(139)Конец сегмента задачи 1
; Сегмент команд задачи 2
text2   segment use16        ;(140)
        assume CS:text2      ;(141)
task2   proc                 ;(142)Процедура задачи 2
        ...    ;Аналогично task1 (другая позиция на экране)
msg2    db 'Работает процедура Task2';(143)Сообщение задачи 2
task2   endp                 ;(144)Конец процедуры задачи 2
text2_size=$-task2           ;(145)Размер сегмента задачи 2
text2   ends                 ;(146)Конец сегмента задачи 2
        end main             ;(147)Конец программы и точка входа
```

Таблица глобальных дескрипторов дополнена двумя новыми дескрипторами сегментов команд gdt_text1 задачи 1 и gdt_text2 задачи 2, имеющими некоторые особенности. Для того чтобы не создавать в каждой задаче собственный сегмент данных, мы поместили контрольные строки текста, выводимые задачами на экран, в их сегменты команд (предложения 136 и 143) – техника, обычная для программ реального режима. Однако обычный сегмент команд с атрибутом 98h можно только исполнять, но не читать. Чтобы разрешить задачам читать данные из их сегментов команд, дескрипторам этих сегментов присвоены атрибуты 9Ah (см. рис. 69.1 и табл. 69.1).

В таблице глобальных дескрипторов появились также три новых дескриптора gdt_tss0, gdt_tss1 и gdt_tss2 сегментов состояния задач TSS. В нашем примере используются TSS минимального размера – по 104 байта, поэтому граница дескрипторов TSS равна 103. Атрибут 1 дескрипторов имеет значение 89h: присутствующий, уровень привилегий DPL=0, свободный TSS (рис. 70.6).



Рис. 70.6. Расшифровка значения атрибута для дескриптора TSS

Вспомним, что дескриптор TSS может быть свободным и занятым (см. табл. 70.1). Описывая сегменты состояния задач в таблице дескрипторов, мы указываем с помощью кода типа, что они свободны. Как только селектор дескриптора TSS будет загружен в регистр задачи TR, процессор изменит код атрибута дескриптора этого TSS, объявив его занятым. При этом TSS исходной задачи остается занятым до ее завершения, даже если происходят переключения на другие задачи. TSS вложенной задачи помечается процессором как занятый, как только произойдет переключение на эту задачу, а после ее завершения и возврата в исходную задачу TSS вложенной задачи снова освобождается. Попытка переключения на задачу, TSS который занят, приводит к исключению нарушения общей защиты, чем предотвращается повторный запуск активной задачи.

В сегменте данных главной задачи зарезервировано место для трех сегментов состояния задач (предложения 26...28). Там же предусмотрены двухсловные поля адресов для команд call dword ptr переключения на вложенные задачи (предложения 29...32). Поскольку переключение осуществляется через TSS задачи, в качестве сегментного адреса задачи указывается селектор ее TSS (в данном случае 64 и 72). Слово со смещением при переключении задач игнорируется, хотя должно присутствовать в программе согласно формату команды call.

Ввиду относительной сложности программы, при отладке которой могут возникать разнообразные ошибки, в программу включены отдельные обработчики исключений: 10 (недопустимый TSS), 11 (ошибка обращения к стеку), 12 (отсутствие сегмента) и 13 (общая защита). Все остальные исключения приводят к вызову общего обработчика dummy.

Процедуры обработчиков исключений несколько изменились. Исключения, как и аппаратные прерывания, возникают асинхронно, в неизвестных заранее точках программы. При передаче управления на обработчик исключения процессор изменяет только содержимое CS:IP (сохраняя вектор прерванной задачи в стеке), но весь контекст задачи, т. е. содержимое сегментных и прочих регистров, остается таким, каким он был в прерванной задаче. Если обработчику исключения требуются какие-либо данные из главной задачи, как это и имеет место в нашей программе, он должен соответствующим образом настроить сегментные регистры. Это, в свою очередь, разрушит контекст прерванной задачи. Поэтому для обработчиков исключений действуют те же правила, что и для обработчиков аппаратных прерываний: при входе в обработчик следует сохранить, а перед завершением восстановить используемые им регистры. Ес-

ли, однако, оформить обработчики как самостоятельные задачи и поместить в дескрипторы исключений селекторы их TSS, сохранение и восстановление контекста прерванной исключением задачи будет выполняться процессором аппаратно в ходе переключения.

В нашем случае все исключения приводят к завершению программного комплекса, поэтому сохранение регистров не требуется, однако настройка сегментного регистра DS для обеспечения обращения к сегменту данных главной задачи необходима. Засылка в DS селектора сегмента data (конкретно числа 8) дает возможность обращаться затем в обработчике к строке string и вызывать подпрограмму wrd_asc, которая также работает с полями данных главной задачи.

Общая структура программы осталась без изменений. На этапе создания таблицы глобальных дескрипторов дополнительно заполняются два дескриптора сегментов команд text1 и text2 (предложения 64...70 и далее), а также три дескриптора, описывающие сегменты состояния задач TSS (предложения 67...72 и далее).

Перед переключением на задачу 1 следует инициализировать некоторые поля TSS1 и TSS2 (предложения 78...89). TSS0 инициализировать не надо, так как он будет заполнен процессором при первом переключении на вложенную задачу. В данном примере инициализация TSS выполняется еще в реальном режиме, хотя эту операцию можно было бы перенести в защищенный режим. Поля для CS и IP TSS1 заполняются селектором сегмента команд задачи 1 и смещением task1 точки входа в задачу 1. Для стека задачи 1 произвольно выделена область начиная с байта 200 нашего сегмента стека (селектор сегмента тот же, а смещение 200; предложения 80 и 81). Поскольку данные задачи 1 в нашем примере расположены в ее сегменте команд, в поле для DS заносится селектор 40 сегмента команд этой задачи (предложение 82). Поле для ES инициализируется селектором видеопамяти (предложение 83). Схожим образом заполняются поля TSS2.

Далее следуют уже рассмотренные ранее операции загрузки регистра IDTR и перехода в защищенный режим.

После перехода в защищенный режим и выполнения обычных процедур инициализации сегментных регистров в регистр задачи TR загружается селектор TSS исходной задачи (предложения 94 и 95), что обеспечит возврат из вложенной задачи в исходную.

Наконец, командами косвенного дальнего вызова (предложения 96 и 97) осуществляется последовательное переключение сначала на задачу 1, а затем на задачу 2. Процедуры обеих задач заканчиваются командами iret переключения на исходную задачу.

В процессе отладки задачи полезно убедиться, что модифицированные процедуры обработчиков исключений работают правильно, выводя на экран диагностическую информацию (номер исключения и адрес его возникновения) как из главной задачи, так и из вложенных задач. Для этого следует включить в программу строки, которые должны заведомо приводить к тому или иному исключению. Например, предложение

```
mov    msg1,AL
```

включенное в любое место процедуры task1, вызовет исключение общей защиты (0Dh), так как при выполнении этой процедуры в регистре DS находится селектор сегмента команд, в который запрещена запись. Точно так же предложения

```
mov    SP,600
pop    AX
```

приведут к исключению нарушения стека (0Ch), поскольку после занесения в регистр SP смещения дна стека извлечь из него уже ничего нельзя.

Следует подчеркнуть, что рассмотренный пример организации переключения задач предназначен лишь для знакомства с элементами архитектуры процессора, а также алгоритмами его работы, связанными с понятием задачи. Вряд ли можно найти какой-то практических смысл в переключении задач нашего примера. Впрочем, это замечание можно отнести к большинству примеров данного раздела. В них иллюстрируются главным образом базовые понятия архитектуры процессора, а отнюдь не принципы построения многозадачных операционных систем защищенного режима.

Статья 71. **Раздельные операционные среды и таблицы локальных дескрипторов**

Задачи task1 и task2 в предыдущем примере хотя и образовывали отдельные сегменты, но, в сущности, не были как-либо защищены друг от друга. Однако наглядно продемонстрировать это обстоятельство не так-то просто. Действительно, понятие защиты обозначает главным образом невозможность одной задаче затереть поля данных другой, а в примере 70.1 задачи task1 и task2 не имеют сегментов данных: выводимые на экран символьные строки расположены в соответствующих сегментах команд. Это, кстати, позволило нам еще раз обратить внимание на свойства сегментов команд в защищенном режиме: сегменты с атрибутом 92h можно только исполнять, сегменты же с атрибутом 9Ah допускают также и чтение (но не запись).

Видоизменим пример 70.1 так, чтобы каждая задача обладала собственными сегментами данных и стека, и покажем незащищенность задач друг от друга (пример 71.1).

Пример 71.1. Задачи с собственными сегментами данных и стека

```
.586P
;Структура для описания дескрипторов сегментов
        ...
;Структура для описания шлюзов ловушек
        ...
;Сегмент данных главной задачи
data    segment use16 ;16-разрядный сегмент
;Таблица глобальных дескрипторов GDT
gdt_null descr <>          ;Селектор 0
gdt_data descr <data_size-1,,,92h>;Селектор 8, сегмент данных
gdt_code descr <code_size-1,,,98h>;Селектор 16, сегмент команд
gdt_stack descr <599,,,92h>;Селектор 24, сегмент стека
gdt_screen descr <3999,8000h,0Bh,92h>;Селектор 32, видеопамять
gdt_text1 descr <text1_size-1,,,98h>; Селектор 40, коды task1
gdt_text2 descr <text2_size-1,,,98h>; Селектор 48, коды task2
gdt_tss0 descr <103,0,0,89h>;Селектор 56, TSS0
gdt_tss1 descr <103,0,0,89h>;Селектор 64, TSS1
gdt_tss2 descr <103,0,0,89h>;Селектор 72, TSS2
gdt_data1 descr <data1_size-1,,,92h>;Селектор 80, данные task1
gdt_data2 descr <data2_size-1,,,92h>;Селектор 88, данные task2
gdt_stk1 descr <255,,,92h> ;Селектор 96, стек task1
gdt_stk2 descr <255,,,92h> ;Селектор 104, стек task2
gdt_size=$-gdt_null        ;Размер GDT
;Таблица дескрипторов прерываний
        ...
```

```
;Различные данные программы
        ...                 ;Как в примере 70.1
;Сегмент команд главной задачи
text    segment use16       ;16-разрядный сегмент
        assume CS:text,DS:data
textseg label word          ;Метка начала сегмента команд
;Обработчики исключений 10, 11, 12, 13 и остальных (dummy)
        ...
main    proc
        xor    EAX,EAX
        mov    AX,data
        mov    DS,AX
;Вычислим и загрузим в GDT линейный адрес сегмента данных data
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента команд text
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента команд text1
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента команд text2
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента стека stk
        ..
;Вычислим и загрузим в GDT линейный адрес сегмента стека task1
        xor    EAX,EAX
        mov    AX,stk1
        shl    EAX,4
        mov    BX,offset gdt_stk1
        mov    [BX].base_l,AX
        shr    EAX,16
        mov    [BX].base_m,AL
;Вычислим и загрузим в GDT линейный адрес сегмента стека task2
        ...
;Вычислим и загрузим в GDT линейные адреса TSS0, TSS1 и TSS2
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента команд data1
        xor    EAX,EAX      ;Очистим EAX
        mov    AX,data1     ;Сегментный адрес сегмента команд
        shl    EAX,4
        mov    BX,offset gdt_data1
        mov    [BX].base_l,AX
        shr    EAX,16
        mov    [BX].base_m,AL
;Вычислим и загрузим в GDT линейный адрес сегмента команд data2
        ...
;Подготовим псевдодескриптор pdescr для загрузки регистра GDTR
        ...
        cli                 ;Запрет прерываний
;TSS0 инициализировать не надо. Инициализируем TSS1
        mov    tss1+4Ch,40 ;CS
        mov    tss1+20h,offset task1;IP
        mov    tss1+50h,96 ;SS
        mov    tss1+38h,256;SP
        mov    tss1+54h,80 ;DS
        mov    tss1+48h,32 ;ES
;Инициализируем TSS2
        mov    tss2+4Ch,48 ;CS
        mov    tss2+20h,offset task2;IP
        mov    tss2+50h,104;SS
        mov    tss2+38h,256;SP
        mov    tss2+54h,88 ;DS
        mov    tss2+48h,32 ;ES
```

```
;Загрузим IDTR
        ...
;Переходим в защищенный режим
        ...
;-------------------------------------------;
; Теперь процессор работает в защищенном режиме ;
;-------------------------------------------;
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:
;Делаем адресуемыми данные и стек, инициализируем ES
        ...
;Загрузим регистр задачи TR селектором TSS главной задачи
        mov    AX,56
        ltr    AX
;Выполним переключение задач
        call   dword ptr task1_offs
        call   dword ptr task2_offs
;Выведем в диагностическую строку код нормального завершения
        ...
;Выведем на экран диагностическую строку
        ...
;Вернемся в реальный режим
;Сформируем и загрузим дескрипторы для реального режима
        ...
;Выполним дальний переход для того, чтобы заново загрузить селектор
;в регистр CS и модифицировать его теневой регистр
        ...
;Переключим режим процессора
        ...
;-----------------------------------------------;
; Теперь процессор снова работает в реальном режиме ;
;-----------------------------------------------;
return:
;Восстановим вычислительную среду реального режима
        ...
;Восстановим состояние регистра IDTR и завершим программу
        ...
main    endp
;Подпрограммы преобразования wrd_asc и bin_asc
        ...
code_size=$-textseg         ;Размер сегмента команд
text    ends
;Сегмент стека главной задачи
stk     segment use16 stack;16-разрядный сегмент
        db     256 dup ('^')
stk     ends
;Сегмент команд задачи 1
text1   segment use16
        assume CS:text1
task1   proc
        mov    AH,1Eh        ;Атрибут
        mov    SI,offset msg1;Смещение строки
        mov    DI,1600       ;Позиция на экране
        mov    CX,24         ;Длина строки
c1:     lodsb                ;Символ в AL
        stosw                ;Символ+атрибут на экран
        loop   c1            ;Цикл
        iret                 ;Завершение задачи 1
task1   endp
```

```
text1_size=$-task1
text1   ends
;Сегмент данных задачи 1
data1   segment use16
msg1    db 'Работает процедура Task1'
data1_size=$-msg1
data1   ends
;Сегмент стека задачи 1
stk1    segment use16 stack
        db 256 dup('1')
stk1    ends
;Сегмент команд задачи 2
        ...    ;Аналогично задаче task, но другая позиция на экране
;Сегмент данных задачи 2
data2   segment use16
msg2    db 'Работает процедура Task2'
data2_size=$-msg2
data2   ends
;Сегмент стека задачи 2
stk2    segment use16 stack
        db 256 dup('2')
stk2    ends
        end main
```

По сравнению с примером 70.1 в таблицу глобальных дескрипторов добавились дескрипторы сегментов данных (gdt_data1, gdt_data2) и стеков (gdt_stk1, gdt_stk2) для задач task1 и task2. Для всех стеков выбран размер 256 байт. Поскольку данные задач теперь будут располагаться в сегментах данных, сегментам команд всех задач присвоен атрибут 98h (только исполнение).

Введение новых дескрипторов потребовало включения в программу блоков вычисления и загрузки в GDT линейных адресов сегментов data1, data2, stk1 и stk2. Несколько изменились блоки инициализации сегментов состояния задач TSS1 и TSS2: в каждом TSS указаны значения собственных селекторов данных и стека и изменены исходные значения SP. Наконец, в задачах task1 и task2 данные перенесены в сегменты данных. В остальном текст программы остался тем же.

Результат выполнения рассмотренной программы ничем не должен отличаться от вывода примера 71.1 (см. рис. 70.4).

Попробуем теперь по ходу выполнения задачи 1 дотянуться до полей данных задачи 2. Сделать это очень просто. Достаточно загрузить в какой-либо сегментный регистр селектор сегмента данных data2; после этого все данные задачи 2 будут доступны задаче 1. Для проверки этого утверждения включим в процедуру task1 следующие строки:

```
push   DS              ;Сохраним DS
mov    AX,88           ;Селектор data2
mov    DS,AX           ;Инициализация DS
mov    BX,0            ;Смещение в data2
mov    word ptr [BX],'**';Затрем часть строки msg2
pop    DS              ;Восстановим DS
```

Вывод видоизмененной программы приведен на рис. 71.1.

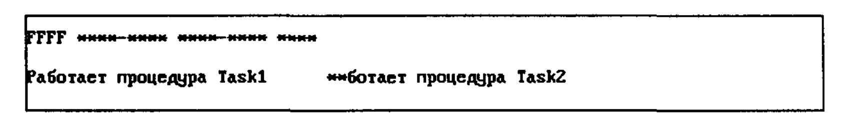


Рис. 71.1. Вывод программы, в которой задача 1 затерла поля данных задачи 2

Для защиты задач друг от друга так же, как и для защиты операционной системы от прикладных задач, в процессоре предусмотрено несколько механизмов защиты. Одним из них является концепция таблиц локальных дескрипторов.

В предыдущих статьях мы познакомились с двумя типами таблиц дескрипторов: таблицей глобальных дескрипторов, в которой описываются сегменты памяти, используемые программой, и таблицей дескрипторов прерываний, которая содержит шлюзы для вызова обработчиков прерываний и исключений. Обе таблицы могут существовать только в одном экземпляре; все задачи (если им это разрешено) могут обращаться к этим таблицам и работать с описанными в них сегментами и программами. Помимо этих таблиц в многозадачной системе можно для каждой задачи построить свою таблицу локальных дескрипторов (Local Descriptor Table, LDT), которая будет определять сегменты памяти, доступные только этой конкретной задаче. Структура таблицы локальных дескрипторов такая же, как у глобальной таблицы, при этом сами локальные таблицы описываются в глобальной таблице с помощью системных глобальных дескрипторов.

Преобразуем пример 71.1, перенеся дескрипторы сегментов задач 1 и 2 из таблицы глобальных дескрипторов GDT в две таблицы локальных дескрипторов LDT. Состав сегментов программного комплекса останется тем же (рис. 71.2); изменению подвергнется состав дескрипторных таблиц и характер взаимодействия элементов комплекса.



Рис. 71.2. Структура программного комплекса примеров 71.1 и 71.2

В действительности наш комплекс имеет значительно больше программных элементов, чем показано на рис. 71.1. В него входят таблицы дескрипторов (глобальных, локальных и прерываний); сегменты состояния задач; обработчики прерываний и исключений; видеопамять. Все эти элементы должны быть описаны в тех или иных таблицах дескрипторов, за исключением "главных" таблиц GDT и IDT, которые не входят в другие таблицы и не имеют соответствующих им дескрипторов.

На рис. 71.3 изображен состав дескрипторных таблиц примера 71.2, позволяющий судить как о детальной структуре программного комплекса, так и о взаимосвязях его отдельных элементов.

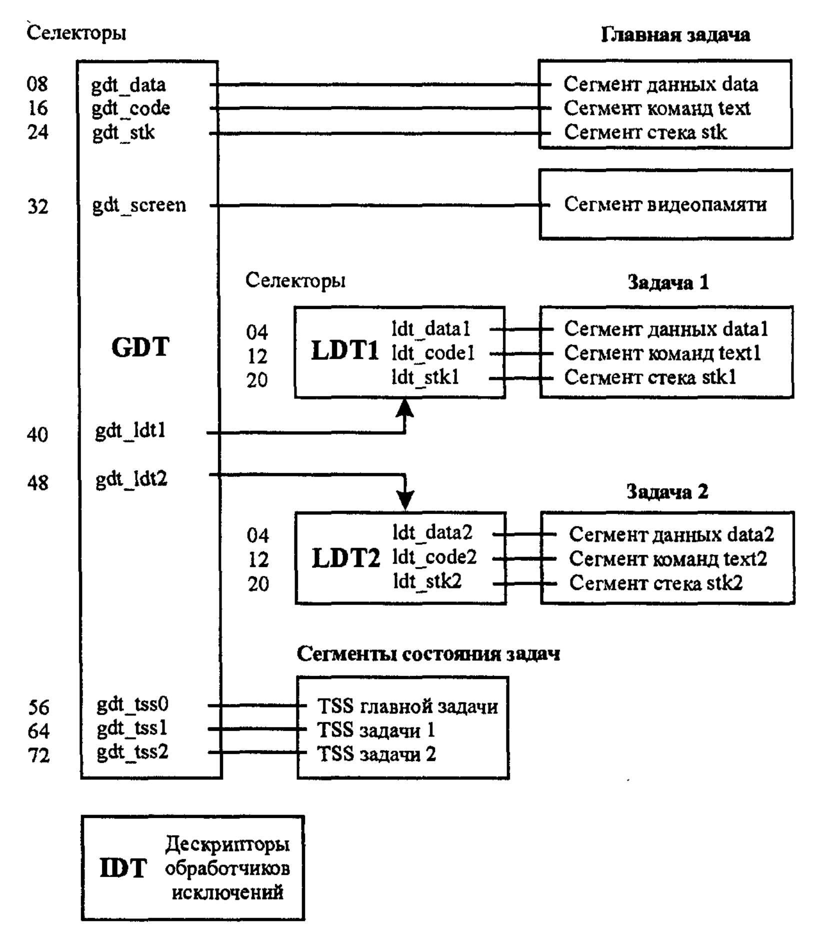


Рис. 71.3. Дескрипторные таблицы и составляющие элементы программного комплекса примера 71.2

Рассмотрим текст программы, которая по своему содержанию и функциям в точности соответствует примеру 71.1, но имеет раздельные операционные среды для каждой задачи (пример 71.2).

Пример 71.2. Переключение задач и локальные таблицы дескрипторов

```
.586P
;Структура для описания дескрипторов сегментов
        ...
;Структура для описания шлюзов ловушек
        ...
;Сегмент данных
data    segment use16 ;16-разрядный сегмент
;Таблица глобальных дескрипторов GDT
gdt_null descr <>          ;Селектор 0
gdt_data descr <data_size-1,,,92h>;Селектор 8, сегмент данных
gdt_code descr <code_size-1,,,98h>;Селектор 16, сегмент команд
gdt_stack descr <255,,,92h>;Селектор 24, сегмент стека
gdt_screen descr <3999,8000h,0Bh,92h>;Селектор 32, видеопамять
```

```
gdt_ldt1 descr <ldt1_size-1,,,82h>;Селектор 40, LDT1
gdt_ldt2 descr <ldt2_size-1,,,82h>;Селектор 48, LDT2
gdt_tss0 descr <103,0,0,89h>;Селектор 56, TSS0
gdt_tss1 descr <103,0,0,89h>;Селектор 64, TSS1
gdt_tss2 descr <103,0,0,89h>;Селектор 72, TSS2
gdt_size=$-gdt_null        ;Размер GDT
;Таблица локальных дескрипторов для задачи 1
ldt1     label word
ldt_data1 descr <data1_size-1,,,92h>;Селектор 4, данные task1
ldt_text1 descr <text1_size-1,,,98h>;Селектор 12, коды task1
ldt_stk1 descr <255,,,92h>;Селектор 20, сегмент стека task1
ldt1_size=$-ldt1
;Таблица локальных дескрипторов для задачи 2
ldt2     label word
ldt_data2 descr <data2_size-1,,,92h>;Селектор 4, данные task2
ldt_text2 descr <text2_size-1,,,98h>;Селектор 12, коды task2
ldt_stk2 descr <255,,,92h>;Селектор 20, сегмент стека task2
ldt2_size=$-ldt2
;Таблица дескрипторов прерываний
         ...
;Различные данные программы
         ...               ;Как в примере 70.1
;Сегмент команд
text     segment use16     ;16-разрядный сегмент
         assume CS:text,DS:data
textseg  label word        ;Метка начала сегмента команд
;Обработчики исключений 10, 11, 12, 13 и остальных (dummy)
         ...
main     proc
         xor    EAX,EAX
         mov    AX,data
         mov    DS,AX
;Вычислим и загрузим в GDT линейный адрес сегмента данных data
         ...
;Вычислим и загрузим в GDT линейный адрес сегмента команд text
         ...
;Вычислим и загрузим в GDT линейный адрес сегмента стека stk
         ...
;Вычислим и загрузим в GDT линейные адреса TSS0, TSS1 и TSS2
         ...
;Вычислим и загрузим в GDT линейный адрес LDT1
         mov    EAX,EBP     ;Линейный адрес сегмента данных
         add    AX,offset ldt1;Добавим смещение LDT1
         mov    BX,offset gdt_ldt1
         mov    [BX].base_l,AX
         shr    EAX,16
         mov    [BX].base_m,AL
;Вычислим и загрузим в GDT линейный адрес LDT2
         ...
;Подготовим псевдодескриптор pdescr для загрузки регистра GDTR
         ...
         cli                ;Запрет прерываний
;Вычислим и загрузим в LDT1 линейный адрес сегмента данных data1
         xor    EAX,EAX
         mov    AX,data1
         shl    EAX,4       ;EAX=линейный базовый адрес
         mov    BX,offset ldt_data1;BX=смещение дескриптора
         mov    [BX].base_l,AX;Загрузим младшую часть базы
         shr    EAX,16      ;Старшую половину EAX в AX
         mov    [BX].base_m,AL;Загрузим среднюю часть базы
;Вычислим и загрузим в LDT1 линейный адрес сегмента команд text1
```

```
        xor    EAX,EAX      ;Очистим EAX
        mov    AX,text1     ;Сегментный адрес сегмента команд
        shl    EAX,4
        mov    BX,offset ldt_text1
        mov    [BX].base_l,AX
        shr    EAX,16
        mov    [BX].base_m,AL
;Вычислим и загрузим в LDT1 линейный адрес сегмента стека stk1
        xor    EAX,EAX
        mov    AX,stk1
        shl    EAX,4
        mov    BX,offset ldt_stk1
        mov    [BX].base_l,AX
        shr    EAX,16
        mov    [BX].base_m,AL
;Вычислим и загрузим в LDT2 линейный адрес сегмента данных data2
        ...
;Вычислим и загрузим в LDT2 линейный адрес сегмента команд text2
        ...
;Вычислим и загрузим в LDT2 линейный адрес сегмента стека stk2
        ...
;TSS0 инициализировать не надо. Инициализируем TSS1
        mov    tss1+4Ch,12;CS, селектор из LDT1
        mov    tss1+20h,offset task1;IP
        mov    tss1+50h,20;SS, селектор из LDT1
        mov    tss1+38h,256 ;SP
        mov    tss1+54h,4;DS, селектор из LDT1
        mov    tss1+48h,32;ES, селектор из GDT
        mov    tss1+60h,40;селектор из LDT1 из GDT
;Инициализируем TSS2
        mov    tss2+4Ch,12;CS
        mov    tss2+20h,offset task2;IP
        mov    tss2+50h,20;SS
        mov    tss2+38h,256;SP
        mov    tss2+54h,4;DS
        mov    tss2+48h,32;ES
        mov    tss2+60h,48;селектор из LDT2 из GDT
;Загрузим IDTR
        ...
;Переходим в защищенный режим
        ...
;-----------------------------------------------;
; Теперь процессор работает в защищенном режиме ;
;-----------------------------------------------;
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:
;Делаем адресуемыми данные и стек, инициализируем ES
        ...
;Загрузим регистр задачи TR селектором TSS главной задачи
        mov    AX,56
        ltr    AX
;Выполним переключение задач
        call   dword ptr task1_offs
        call   dword ptr task2_offs
;Выведем в диагностическую строку код нормального завершения
        ...
;Выведем на экран диагностическую строку
        ...
;Вернемся в реальный режим
;Сформируем и загрузим дескрипторы для реального режима
```

```
        ...
;Выполним дальний переход для того, чтобы заново загрузить селектор
;в регистр CS и модифицировать его теневой регистр
        ...
;Переключим режим процессора
        ...
;------------------------------------------------;
; Теперь процессор снова работает в реальном режиме ;
;------------------------------------------------;
return:
;Восстановим вычислительную среду реального режима
        ...
;Восстановим состояние регистра IDTR и завершим программу
        ...
main    endp
;Подпрограммы преобразования wrd_asc и bin_asc
        ...
code_size=$-textseg          ;Размер сегмента команд
text    ends
;Сегмент стека главной задачи
        ...
;Сегмент команд задачи 1
text1   segment use16
        assume CS:text1
task1   proc
        ...
task1   endp
text1_size=$-task1
text1   ends
;Сегмент данных задачи 1
data1   segment use16
msg1    db 'Работает процедура Task1'
data1_size=$-msg1
data1   ends
;Сегмент стека задачи 1
stk1    segment use16 stack;16-разрядный сегмент
        db 256 dup ('1')
stk1    ends
;Сегмент команд задачи 2
        ...
;Сегмент данных задачи 2
        ...
;Сегмент стека задачи 2
        ...
        end main
```

В таблицу глобальных дескрипторов по-прежнему входят дескрипторы, описывающие сегменты данных, команд и стека главной задачи, а также сегмент видеопамяти и три сегмента состояния задач. Кроме этого, в GDT включены два системных дескриптора, описывающих две таблицы локальных дескрипторов LDT для задач 1 и 2. Сами эти таблицы в нашем примере находятся в сегменте данных главной задачи, хотя их можно выделить в самостоятельные сегменты.

Таблицу локальных дескрипторов описывает дескриптор системного сегмента, имеющий такой же формат, что и дескриптор TSS, и отличающийся от последнего только полем типа (в четырех младших битах байта атрибута 1 записывается код 2).

Поле границы дескрипторов LDT определяется из фактического размера LDT, а атрибут 1 имеет значение 82h: присутствующий, уровень привилегий DPL=0, дескриптор LDT (рис. 71.4).

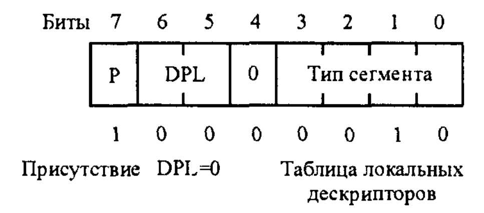


Рис. 71.4. Расшифровка значения атрибута для дескриптора LDT

Каждая из двух имеющихся в программе таблиц локальных дескрипторов включает три дескриптора сегментов памяти задачи: данных, команд и стека. Сами сегменты описаны в конце текста программы.

Место дескриптора в дескрипторной таблице определяет селектор, с помощью которого программа обращается к данному сегменту (см. рис. 65.2).

Поле RPL (Requested Privilege Level, запрошенный уровень привилегий), занимающее биты 0...1 селектора, может принимать значение от 0 до 3, в соответствии с числом уровней привилегий процессора. Назначение этого поля будет объяснено в следующей статье; пока примем его равным нулю.

Бит TI (Table Indicator, индикатор таблицы) равен нулю, если селектор относится к глобальной таблице, и единице, если соответствующий селектору дескриптор входит в локальную таблицу. В битах 3...15 располагается порядковый номер (индекс) дескриптора в таблице дескрипторов. Индексы нумеруются с нуля: 0, 1, 2 и т. д.

Сегменты, описанные в таблице глобальных дескрипторов, имели в наших программах селекторы, численно кратные восьми: 0, 8, 16, 24 и т. д. Для дескрипторов, входящих в LDT, селекторы будут иметь установленным бит 2 и, соответственно, иметь значения 4, 12 и 20. При этом селекторы обеих LDT совпадают. Это не приведет к путанице, так как каждая задача будет обращаться (с помощью селекторов) только к своей локальной таблице, а под одинаковыми индексами в разных таблицах описаны физически разные сегменты.

Введение двух таблиц локальных дескрипторов требует вычисления и занесения в соответствующие дескрипторы, на этапе инициализации, во-первых, линейных адресов самих локальных таблиц (в GDT) и, во-вторых, линейных адресов всех локальных сегментов (в обеих LDT).

Как и раньше, TSS главной задачи не нуждается в инициализации; он будет автоматически заполнен контекстом главной задачи при переходе на задачу 1 или 2. В поля для сегментных регистров в TSS1 и TSS2 заносятся селекторы дескрипторов из локальных таблиц (численно одинаковые для обеих задач). В каждом TSS необходимо заполнить поле со смещением 60h для селектора "своей" таблицы локальных дескрипторов LDT1 и LDT2. Эти селекторы (40 и 48 в примере) соответствуют дескрипторам GDT, описывающим местоположение и другие характеристики локальных таблиц.

Тексты процедур task1 и task2 не изменились, хотя работать они будут по-другому: каждая со своим стеком и своим сегментом данных. Внешне порядок работы с программным комплексом остался тем же: после запуска начинает работать главная

задача, которая после перехода в защищенный режим последовательно переключается на задачи 1 и 2, которые выводят на экран диагностические строки (см. рис. 70.4).

В рассмотренном примере уже не удастся из задачи 1 обратиться к полям данных задачи 2. Действительно, загрузка в сегментный регистр любого из локальных селекторов 4, 12 или 20 адресует нас к полям той же самой задачи. Для того чтобы загрузить селектор сегмента данных другой задачи, как это мы сделали в примере 71.1, необходимо сменить таблицу локальных дескрипторов, а это действие выполняется только при переключении задач. Таким образом, использование локальных адресных пространств привело к надежной защите задач task1 и task2 друг от друга. Подчеркнем, что эта защита реализуется не какими-то программными ухищрениями управляющей задачи или операционной системы, а алгоритмами смены задач, заложенными в архитектуру процессора.

Между прочим, в нашем весьма примитивном примере главная, управляющая задача никак не защищена от задач task1 и task2, поскольку ее сегменты находятся в глобальном адресном пространстве и, следовательно, доступны всем составляющим программного комплекса. Обе задачи вполне могут загрузить в какой-либо сегментный регистр селектор сегмента данных главной задачи 8 и сколько угодно затирать ее поля данных. Убедиться в этом можно, включив в процедуру task1 строки

```
mov   AX,8          ;Селектор данных главной задачи
mov   FS,AX         ;Свободный сегментный регистр
mov   FS:string+25,'!'
```

После завершения программы на экран будет выведена строка

FFFF ****-**** ****-**** !***

Защита управляющих программ от прикладных обычно осуществляется с помощью другого механизма защиты, также встроенного в архитектуру процессора, именно системы уровней привилегий. Этот вопрос будет рассмотрен в следующей статье.

Статья 72. **Уровни привилегий и защиты по привилегиям**

Выделение для каждой задачи с помощью таблицы локальных дескрипторов отдельного локального адресного пространства позволяет надежно защитить задачи друг от друга. Поскольку в процессоре имеется только один регистр LDTR, в каждый момент доступна только одна таблица локальных дескрипторов и, соответственно, только одно локальное адресное пространство. Поэтому каждая задача работает только со своими сегментами, не имея доступа к сегментам других задач. С другой стороны, сегменты, описанные в GDT, доступны всем задачам, которые могут как обращаться к глобальным полям данных, так и вызывать подпрограммы, входящие в глобальные сегменты команд. Концепция локальных дескрипторов позволяет выделять каждой задаче любые ресурсы памяти, надежно защищенной на аппаратном уровне от поползновений других задач.

Учитывая исключительную важность защиты системных ресурсов в многозадачном режиме, в процессорах Intel предусмотрено еще два механизма защиты: страничная организация памяти и система уровней привилегий. В настоящей статье рассматриваются основы защиты задач с помощью уровней привилегий. Страничному преобразованию будет посвящена последняя статья этого раздела.

Каждому сегменту программы придается определенный уровень привилегий, указываемый в поле DPL (Descriptor Privilege Level, уровень привилегий дескриптора) его дескриптора. Уровни привилегий указываются во всех дескрипторах: памяти, системных и шлюзах. Уровень привилегий, указанный в дескрипторе, назначается всем объектам, входящим в данный сегмент. Так, если сегменту команд назначен уровень 3, то все процедуры этого сегмента имеют уровень защиты 3. Точно так же все данные, входящие в сегмент данных, имеют тот уровень защиты, который указан в дескрипторе этого сегмента.

Уровень привилегий выполняемого в данный момент сегмента команд называется текущим уровнем привилегий – CPL (Current Privilege Level). Он определяется полем RPL селектора сегмента команд (см. рис. 65.2), загружаемого в CS. Вся система привилегий основана на сравнении CPL выполняемой программы с уровнями привилегий DPL сегментов, к которым она обращается.

Всего процессор различает 4 уровня привилегий от 0 (максимальные привилегии) до 3 (минимальные). Чем больше численное значение уровня привилегий, тем меньшими привилегиями обладает данный сегмент. Для того чтобы избежать двусмысленности при сравнении привилегий сегментов, мы будем преимущественно называть уровни с большими привилегиями внутренними, с меньшими – внешними. Эти определения связаны с принятым изображением уровней в виде концентрических колец, называемых кольцами защиты (рис. 72.1). Во внешнем кольце располагаются сегменты, которым присвоен уровень 3; ближе к центру располагаются кольца с большими привилегиями и меньшими численными значениями уровней защиты.



Рис. 72.1. Уровни привилегий и кольца защиты

В кольце 0 с наивысшим уровнем привилегий разумно располагать наиболее ответственную часть операционной системы – ее ядро; в кольцах 1 и 2 могут располагаться остальные программы операционной системы и программного обеспечения; наконец, наименее привилегированное кольцо 3 отдается прикладным программам пользователей. Часто применяются только два уровня – 0 для операционной системы и 3 для прикладных программ.

Основные правила защиты по привилегиям сводятся к следующим:

- данные из сегмента с некоторым уровнем привилегий могут быть получены только программой того же или более внутреннего уровня;
- процедура из сегмента с некоторым уровнем привилегий может быть вызвана только программой того же или более внешнего уровня, причем такой вызов должен осуществляться не непосредственно, а через шлюз вызова;
- каждый уровень привилегий имеет свой стек, поэтому при переключении на другой уровень не может произойти разрушения исходного стека;
- все команды, воздействующие на механизмы сегментации и защиты (lgdt, lidt, lldt, ltr и ряд других), являются привилегированными и могут использоваться только в процедурах уровня 0, что исключает вмешательство в организацию программного комплекса со стороны прикладных программ.

Защита по уровням привилегий и защита с помощью локальных адресных пространств действуют параллельно и в какой-то мере независимо. В примере 71.1 мы защитили прикладные задачи, выделив каждой из них свое локальное адресное пространство, однако защита по уровням привилегий в этом примере отсутствовала, так как уровни привилегий всех дескрипторов комплекса были равны нулю (рис. 72.2).

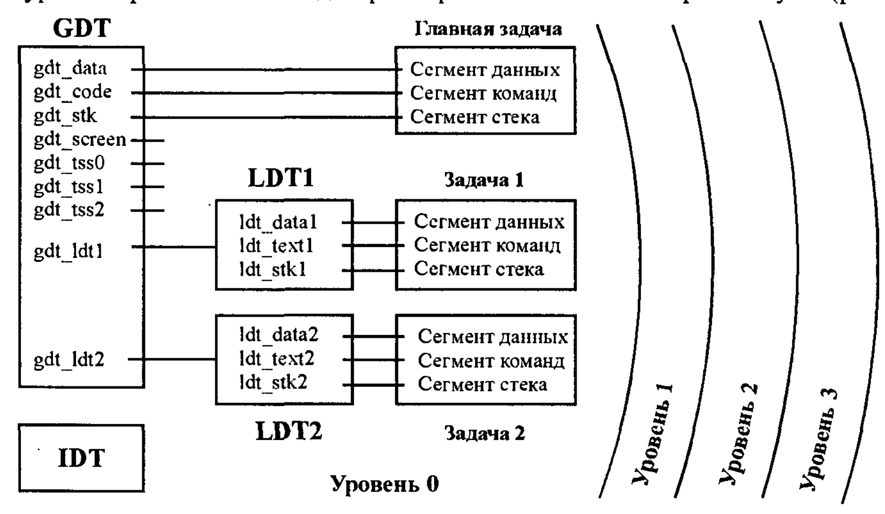


Рис. 72.2. Размещение всего программного комплекса в кольце 0

В результате прикладные задачи, входящие в комплекс, могли бы, например, изменить содержимое глобальных или локальных дескрипторных таблиц, т. е. вмешаться в саму организацию комплекса. Если же разместить сегменты прикладных программ на более внешнем уровне, например уровне 3 (рис. 72.3), они, во-первых, потеряют права на выполнение привилегированных команд и, во-вторых, смогут обращаться только к тем процедурам управляющей программы, которым соответствуют шлюзы вызова, т. е. к процедурам, специально предназначенным для вызова из внешних колец. К полям данных управляющей программы, в частности к таблицам дескрипторов или сегментам состояния задач, доступ из прикладных программ будет закрыт.

Еще одним элементом организации программного комплекса, действующим независимо как от уровней привилегий, так и от разделения адресного пространства на глобальные и локальные области, является выделение системных или прикладных программ в отдельные задачи (именно этот случай изображен на рис. 72.3). В примере 71.2 процедуры, входящие в программу, образовывали отдельные задачи со своими сегментами состояния задач, что обеспечило возможность использования удобных средств переключения задач с автоматическим сохранением их контекстов. Однако разделение на задачи не обязательно. Все программные элементы комплекса, расположенные, возможно, на разных уровнях привилегий и работающие в различных адресных пространствах (глобальном и локальных), могут входить в единую задачу. При этом организация комплекса упрощается, хотя снижается его гибкость.

При построении многоуровневой системы особую сложность представляет организация передачи управления с уровня на уровень. Как уже отмечалось, обычно в кольце 0 располагают программы операционной системы, управляющие вычисли-

тельным процессом; в кольце 3 размещаются прикладные программы. В этом случае сначала начинает работать управляющая программа, которая готовит операционную среду защищенного режима для себя и прикладных программ, после чего переводит процессор в защищенный режим. Затем управление передается прикладной программе, которая часть работы может выполнять самостоятельно, однако должна иметь возможность обращаться к операционной системе для вызова определенных системных функций, например вывода на экран или в файлы, завершения работы и передачи управления системе и т. д. С другой стороны, программы реализации тех системных функций, которые должны вызываться из прикладных задач, могут быть расположены в кольце 3; тогда для их вызова не потребуется переход на внутренний уровень.

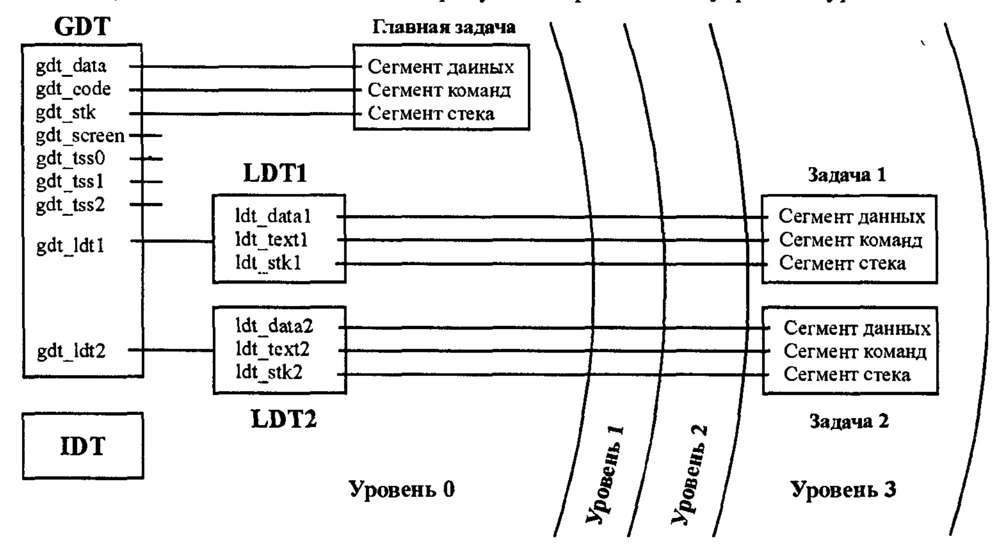


Рис. 72.3. Размещение управляющей программы в кольце 0, а прикладных – в кольце 3

Для того чтобы разобраться в не очень простой технике передачи управления процедурам других уровней, рассмотрим сначала пример однозадачного программного комплекса, в котором все программные элементы входят в одну-единственную задачу. Наш комплекс содержит управляющую программу, имитирующую некоторые функции операционной системы, и прикладную программу, для которой выделяется локальное адресное пространство. Управляющая программа вместе со всеми "системными" областями (таблицами дескрипторов, процедурами обработки исключений и пр.) располагается на уровне 0. Прикладная программа, содержащая сегменты команд, данных и стека, вынесена на уровень 3. Будучи инициирована управляющей программой, она выводит на экран контрольное сообщение и обращается к двум сервисным функциям управляющей программы. Первая функция является упрощенным аналогом функции 13h прерывания 10h системы BIOS (вывод на экран строки); вторая функция позволяет завершить прикладную программу и вернуться в управляющую программу (в MS-DOS эти действия выполняет функция 4Ch).

Пример 72.1. Межуровневая передача управления

```
.586P
;Структура для описания дескрипторов сегментов
```

```
        ...
;Структура для описания шлюзов ловушек
        ...
;Сегмент данных управляющей программы
data    segment use16
;Таблица глобальных дескрипторов GDT
gdt_null descr <>         ;Селектор 0
gdt_data descr <data_size-1,,,92h>;Селектор 8 - сегмент данных
gdt_code descr <code_size-1,,,9Ah>;Селектор 16 - сегмент команд
gdt_stk descr <255,,,92h>;Селектор 24 - сегмент стека
gdt_screen descr <3999,8000h,0Bh,0F2h,> ;Селектор 32 - видеопамять
gdt_tss descr <103,,,89h>;Селектор 40 - дескриптор TSS
gdt_ldt descr <ldt_size-1,,,82h>;Селектор 48 - дескриптор LDT
gdt_size=$-gdt_null        ;Размер GDT
;Таблица локальных дескрипторов. У всех дескрипторов DPL=3
ldt_data descr <data1_size-1,0,0,0F2h>;Селектор 0+4, сегмент данных
ldt_code descr <text1_size-1,0,0,0FAh,0,0>;Селектор 8+4+3 - сегмент команд
ldt_stk descr <255,0,0,0F2h,0,0>;Селектор 16+4+3 - сегмент стека
ldt_gate1 trap <srv1,16,2,0ECh,0>;Селектор 24+4 - процедура srv1, счетчик=2
ldt_gate2 trap <srv2,16,0,0ECh,0>;Селектор 32+4 - процедура srv2
ldt_size=$-ldt_data
;Таблица дескрипторов прерываний
        ...
;Различные данные программы
pdescr  df 0               ;Псевдодескриптор
msgctl  db 'Управляющая программа начала работу'
msgctl_len=$-msgctl
string  db '**** ****-**** ****-**** ****'
;           0    5    10   15   20   25
len=$-string
msg     db 27,'[31;42m  Вернулись в реальный режим  ',27,'[0m$'
tss     dw 52 dup (0)      ;TSS задачи
data_size=$-gdt_null       ;Размер сегмента данных
data    ends
text    segment use16
        assume CS:text,DS:data
textseg label word         ;Метка начала сегмента команд
;Обработчики исключений 10, 11, 12, 13 и остальных (dummy)
        ...
;Сервисная функция 1
srv1    proc   far         ;Дальная процедура
        mov    EBP,ESP     ;Текущий указатель стека
        add    EBP,8       ;Сместимся к началу параметров
        mov    SI,[BP]+0   ;Относительный адрес строки в ESI
        mov    AX,[BP]+2   ;Сегмент строки в AX
        mov    DS,AX       ;Настроим DS на сегмент строки
        mov    CX,[BP]+4   ;Длина строки в CX
        mov    DI,[BP]+6   ;Позиция на экране в DI
        cld                ;Вперед
        mov    AH,14h      ;Атрибут
scr1:   lodsb              ;Получим байт строки
        stosw              ;Байт с атрибутом на экран
        loop   scr1
        db     66h         ;Возврат со снятием со стека
        ret    8           ;2 двойных слов
srv1    endp
;Сервисная функция 2
srv2    proc   far         ;Дальняя процедура
        mov    AX,8        ;Восстановим адресуемость данных
        mov    DS,AX       ;управляющей программы
        mov    SP,256      ;Восстановим указатель стека
```

```
        mov     EAX,5F4B5F4Fh;Коды 'OK' с атрибутами
        mov     ES:[2880],EAX;На экран
        jmp     fin ;Переход в управляющую программу (на завершение)
srv2    endp
main    proc
        xor     EAX,EAX
        mov     AX,data
        mov     DS,AX
;Вычислим и загрузим в GDT линейный адрес сегмента данных data
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента команд text
        ...
;Вычислим и загрузим в GDT линейный адрес сегмента стека stk
        ...
;Вычислим и загрузим в GDT линейный адрес TSS
        ...
;Вычислим и загрузим в GDT линейный адрес LDT
        ...
;Вычислим и загрузим в LDT линейный адрес сегмента данных data1
        ...
;Вычислим и загрузим в LDT линейный адрес сегмента команд text1
        ...
;Вычислим и загрузим в LDT линейный адрес сегмента стека stk1
        ...
;Подготовим псевдодескриптор pdescr для загрузки регистра GDTR
        ...
        cli                  ;Запрет прерываний
;Загрузим IDTR
        ...
;Переходим в защищенный режим
        ...
;------------------------------------------;
; Теперь процессор работает в защищенном режиме ;
;------------------------------------------;
;Загружаем в CS:IP селектор:смещение точки continue
        ...
continue:
;Делаем адресуемыми данные и стек, инициализируем ES
        ...
;Выведем на экран сообщение главной процедуры
        mov     CX,msgctl_len
        mov     SI,offset msgctl
        mov     AH,3Eh
        mov     DI,1920
scr2:   lodsb
        stosw
        loop    scr2
;Подготовим вызов прикладной программы
;Инициализируем TSS
        mov     dword ptr tss+4,ESP ;ESP0
        mov     tss+8,24    ;SS0
;Загрузим регистр LDTR адресом LDT
        mov     AX,48
        lldt    AX           ;Загрузим регистр LDTR
;Загрузим TR селектором TSS
        mov     AX,40
        ltr     AX
;Подготовим стек для команды retf перехода на прикладную программу
        mov     EAX,23      ;SS
        push    EAX
        mov     EAX,256     ;ESP
```

```
        push   EAX
        mov    EAX,15       ;CS
        push   EAX
        mov    AX,offset appl ;EIP
        push   EAX
        db     66h          ;"Возврат из подпрограммы"
        retf                ;в прикладную программу
fin:    mov    AX,0FFFFh    ;Диагностическое значение
home:   mov    SI,offset string
        call   wrd_asc
;Выведем на экран диагностическую строку
        ...
;Вернемся в реальный режим
;Сформируем и загрузим дескрипторы для реального режима
        ...
;Выполним дальний переход для модификации теневого регистра CS
        ...
;Переключим режим процессора
        ...
;-----------------------------------------------------;
; Теперь процессор снова работает в реальном режиме ;
;-----------------------------------------------------;
return:
;Восстановим вычислительную среду реального режима
        ...
;Восстановим состояние регистра IDTR реального режима и завершим программу
        ...
        main   endp
;Подпрограммы преобразования числа в символьную форму wrd_asc и bin_asc
        ...
code_size=$-textseg
text    ends
;Сегмент стека управляющей программы
        ...
;Сегмент данных прикладной программы
data1   segment use16
msg1    db 'Вызвана прикладная программа'
msg1_len=$-msg1
msg2    db 'Вызвана сервисная функция 1 управляющей программы'
msg2_len=$-msg2
data1_size=$-msg1
data1   ends
;Сегмент команд прикладной программы
text1   segment use16
        assume CS:text1,DS:data1
appl    proc
        mov    AX,4         ;Инициализируем DS
        mov    DS,AX
;Выведем на экран контрольное сообщение
a1:     mov    AH,1Bh       ;Атрибут
        mov    DI,2240      ;Позиция на экране
        mov    CX,msg1_len;Длина строки
        mov    SI,offset msg1;Смещение на экране
        cld
scr4:   lodsb
        stosw
        loop   scr4
;Вызовем сервисную функцию 1 управляющей программы для вывода на экран
;сообщения "средствами OS". Подготовим в стеке параметры (2 двойных слова)
        mov    AX,2560      ;Позиция на экране
```

```
        shl     EAX,16      ;Сдвинем в старшую половину EAX
        mov     AX,msg2_len ;Длина сообщения
        push    EAX         ;Двойное слово в стек
        mov     AX,DS       ;Сегмент строки
        shl     EAX,16      ;Сдвинем в старшую половину EAX
        mov     AX,offset msg2;Смещение строки
        push    EAX         ;Двойное слово в стек
        db      9Ah         ;Код команды call far ptr
        dw      0           ;Игнорируемое смещение точки вызова
        dw      28          ;Селектор шлюза вызова процедуры srv1 в LDT
;Вызовем сервисную функцию 2 управляющей программы (завершение
;всей программы). Она не требует параметров в стеке
        db      9Ah         ;Код команды call far ptr
        dw      0           ;Игнорируемое смещение точки вызова
        dw      36          ;Селектор шлюза вызова процедуры srv2 в LDT
appl    endp
text1_size=$-appl
text1   ends
;Сегмент стека прикладной программы
stk1    segmentuse16
        db      256 dup ('T')
stk1    ends
        end     main
```

На рис. 72.4 изображена структурная схема программного комплекса примера 72.1 с указанием состава дескрипторных таблиц, численных значений селекторов сегментов и распределения сегментов по кольцам защиты.

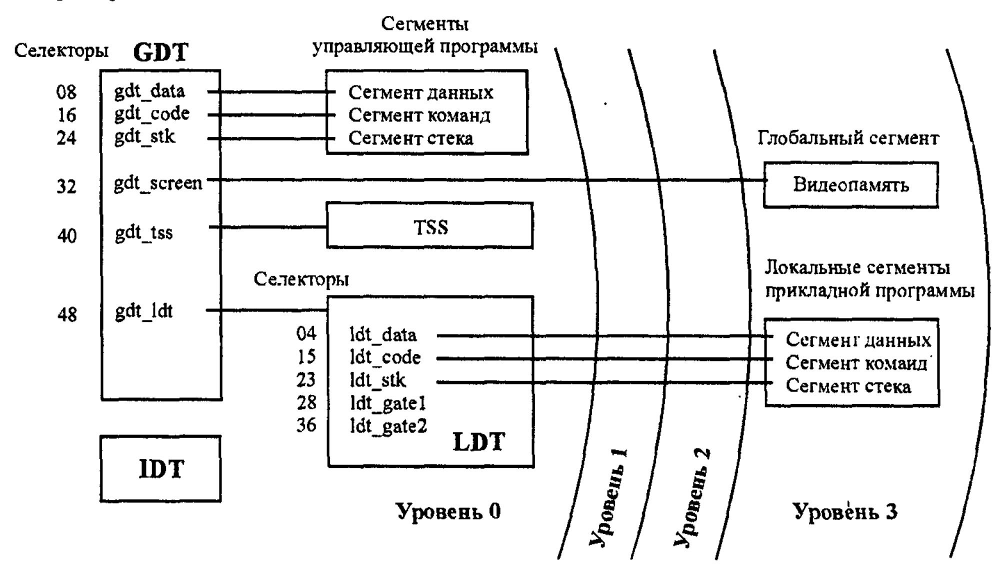


Рис. 72.4. Структурная схема программного комплекса примера 72.1

В таблицу глобальных дескрипторов включены дескрипторы всех трех сегментов управляющей программы, а также сегментов видеопамяти, состояния задачи и таблицы локальных дескрипторов. Сегменты программы, TSS и LDT имеют в байте атрибутов, как и раньше, значение поля DPL, равное нулю, и располагаются, таким образом, в нулевом, наиболее привилегированном кольце защиты. В дескрипторе

видеопамяти поле DPL содержит число 3 (рис. 72.5), т. е. сегмент видеопамяти расположен на уровне 3 с минимальными привилегиями. Это разрешает доступ к нему из программ всех уровней. Для того чтобы сделать обращение к видеопамяти практически возможным, он описан в таблице глобальных дескрипторов и размещен, таким образом, в глобальном адресном пространстве.

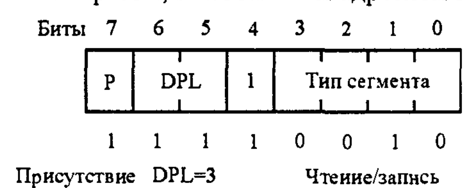


Рис. 72.5. Расшифровка значения атрибута сегмента видеопамяти

В единственной таблице локальных дескрипторов описаны сегменты данных, команд и стека прикладной программы, а также два шлюза вызова процедур управляющей программы ldt_gate1 и ldt_gate2. Все сегменты прикладной программы, как и видеопамять, имеют DPL=3 и размещаются на уровне 3.

Шлюзы вызова, описывающие процедуры srv1 и srv2, содержат относительные адреса точек входа в эти процедуры (вообще говоря, 32-битовые, но в нашем случае фактически 16-битовые), селектор сегмента команд управляющей программы (16), поле счетчика (2 для первого шлюза и 0 для второго), а также байт атрибута, равный ECh (рис. 72.6).

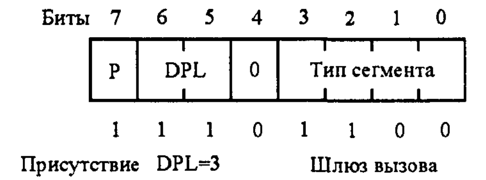


Рис. 72.6. Расшифровка значения атрибута шлюза вызова

На первый взгляд кажется, что DPL шлюза, описывающего процедуру кольца 0, должен быть равен нулю, однако для шлюзов вызова действуют особые правила назначения уровня привилегий. DPL шлюза определяет не уровень привилегий вызываемой процедуры, а то, какими привилегиями должна обладать вызывающая программа. Поскольку мы будем вызывать процедуры srv1 и srv2 из кольца 3, то и дескриптор шлюза должен иметь DPL, не меньший 3, т. е. просто 3.

Назначение поля счетчика будет разъяснено позже.

Как уже отмечалось, наш программный комплекс, несмотря на наличие в нем нескольких сегментов команд, расположенных к тому же на разных уровнях привилегий, представляет собой единую задачу. Поэтому в сегменте данных управляющей программы зарезервировано поле для одного сегмента состояния задачи tss. Заполнен этот сегмент будет позже, по ходу выполнения программы.

В сегмент данных включена символьная строка mesctrl для контрольного сообщения управляющей программы. Остальные поля данных были описаны в предыдущих примерах.

Сервисная функция 1, посылающая на экран строку текста, требует для своей работы ряда параметров, передаваемых ей через стек. Правила передачи параметров будут рассмотрены позже.

Сервисная функция 2 предназначена для завершения работы прикладной программы и возврата в управляющую. В процедуре srv2 заново инициализируются

DS и SP, на экран выводится строка 'OK' и управление передается на завершение всего программного комплекса (метка fin).

Главная процедура main, олицетворяющая управляющую программу, прежде всего выполняет обычные действия по заполнению дескрипторных таблиц GDT и LDT, загрузке регистров GDTR и IDTR и переходу в защищенный режим.

В защищенном режиме инициализируются регистры CS (с помощью команды far jmp), DS, SS и ES, на экран выводится контрольное сообщение, после чего управляющая программа приступает к подготовке перехода на выполнение прикладной программы. Сам вызов прикладной программы осуществляется, как это ни странно, командой ret. Для того чтобы понять, как команда ret может вызвать на выполнение процедуру, придется подробно рассмотреть механизм межуровневого вызова подпрограмм с помощью команды call и шлюза вызова.

Передача управления в заданную точку программы в пределах одного уровня осуществляется с помощью команд call и jmp, причем при переходе в тот же сегмент команд используются ближние формы этих команд, а при переходе в другой сегмент – дальние. Переход на другой уровень возможен только с помощью команды дальнего вызова call; команда jmp передать управление на другой уровень не может. В качестве аргумента команды call должен использоваться не фактический адрес вызываемой процедуры, а селектор шлюза вызова, который, в свою очередь, содержит адрес вызываемой процедуры.

При вызове процедуры через шлюз вызова в поле сегмента указывается, как это и положено в защищенном режиме, селектор шлюза, а поле смещения процессором игнорируется и может содержать любое значение.

В шлюзе вызова (см. рис. 67.1) указывается полный адрес (селектор + смещение) вызываемой процедуры, уровень привилегий шлюза, назначение которого отмечалось выше, и (в поле счетчика) число двухсловных параметров, передаваемых вызываемой процедуре.

Вызов процедуры внутреннего кольца через шлюз вызова не только передает управление на вызываемую процедуру со сменой текущего уровня привилегий, но и копирует указанное в шлюзе число двухсловных параметров из стека внешнего уровня на стек внутреннего. Состояние стека (SS:ESP) внутреннего уровня берется процессором из сегмента состояния задачи. Таким образом, сегмент состояния задачи потребовался в данном примере не ради переключения на другую задачу (у нас все компоненты программы входят в одну задачу), а из-за того, что мы хотим перейти на другой уровень привилегий, что требует смены стека. Перед вызовом процедуры внутреннего уровня следует, во-первых, заполнить в TSS задачи поля для SS и ESP тех внутренних уровней, процедуры которых предполагается вызывать через шлюзы вызова, и, во-вторых, передать процессору селектор TSS (с помощью команды ltr).

В нашем примере для привилегированных программ используется только одно, нулевое кольцо, поэтому в TSS заполняются лишь поля для SS:ESP уровня 0. Если планируется вызывать процедуры из колец 1 или 2, в TSS следует записать исходные значения SS:ESP и для этих уровней (см. формат TSS на рис. 70.3).

Процессор, инициализировав стек внутреннего уровня с помощью данных, полученных из TSS, сохраняет в новом стеке значения SS и ESP, чтобы иметь возможность после завершения внутренней процедуры переключиться на стек внешнего уровня. Далее в стек внутреннего уровня из внешнего стека копируются двухсловные

параметры, число которых (от 0 до 31) задано в шлюзе вызова. Наконец, во внутренний стек помещается, как это всегда делается при дальнем вызове, адрес возврата в вызывающую программу внешнего уровня (значения CS:EIP). В результате указатель стека внутреннего уровня оказывается смещенным относительно первоначального положения на p*4+16 байт, где p – число передаваемых параметров. На рис. 72.7 изображены оба стека, участвующие в операции вызова процедуры внутреннего уровня с передачей двух параметров.

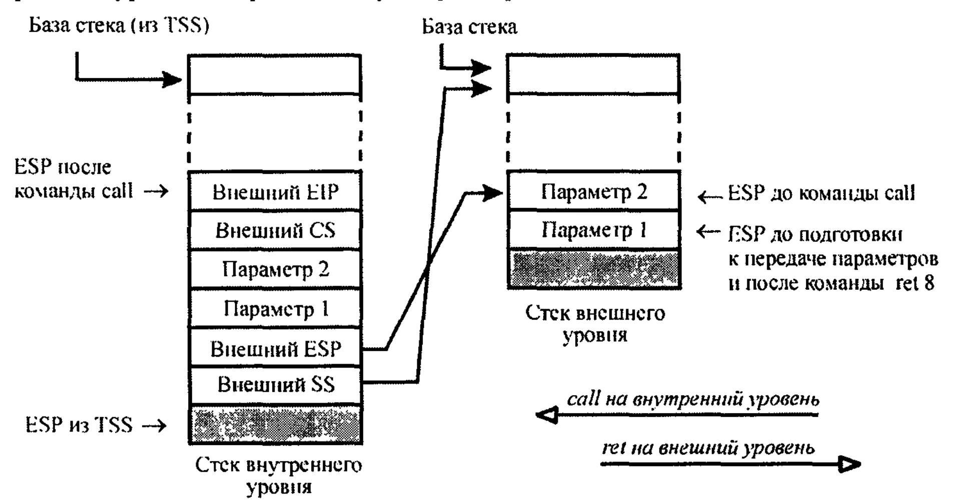


Рис. 72.7. Вызов процедуры внутреннего уровня с передачей параметров через стек

Таким образом, вызывающая программа внешнего уровня должна поместить в стек заданное число двухсловных параметров и выполнить команду дальнего вызова с указанием в качестве сегментного адреса перехода селектор шлюза вызова.

В рассматриваемом примере процедура srv1 должна получить следующие данные: позицию выводимой строки, ее длину, а также полный адрес этой строки в программе (селектор + смещение). Мы упаковали эти данные в два двойных слова, как это показано на рис. 72.8.

Старшая половина	Младшая половина	
Позиция строки на экране	Длина строки	Параметр 1
Селектор сегмента данных	Смещение строки	Параметр 2

Рис. 72.8. Состав параметров для процедуры srv1

В вызываемой процедуре внутреннего уровня srv1 параметры не следует снимать со стека, чтобы не потерять адрес возврата. Извлечение из стека параметров (лучше сказать, чтение параметров) осуществляется стандартным образом с помощью регистра EBP, настраиваемого на текущую вершину стека. В нашем случае параметр 2 оказывается "погребенным" в стеке на глубине 8 байт. Соответственно в процедуре srv1 для адресации этого параметра к текущему указателю стека после переноса его в базовый регистр EBP прибавляется 8. Извлечение передаваемых данных из стека осуществляется с учетом их расположения в двойных словах, как это показано на рис. 72.8.

Команда дальнего возврата ret передает управление в вызвавшую процедуру. Для того чтобы в процессе возврата со стеков были сняты параметры, в качестве аргумента команды ret следует указать суммарное число байтов в параметрах (у нас 8). Команда ret должна снимать со стека двойные слова, поэтому ей предпослан префикс 66h замены размера операнда.

Вторая сервисная функция управляющей программы srv_2 не требует параметров. Соответственно поле для счетчика в ее шлюзе содержит 0, при вызове этой процедуры параметры в стек не заносятся, а возврат должен осуществляться командой ret без параметров. В нашем примере возврат из процедуры srv_2 не предусмотрен.

Рассмотрим оставшийся неясным вопрос, как из управляющей программы, работающей на уровне привилегий 0, перейти в прикладную программу. Как уже отмечалось, команда call допускает переход только на внутренний уровень, но не на внешний. Однако команда ret как раз, наоборот, позволяет вернуться на внешний уровень. Этой командой можно воспользоваться и для первого перехода в прикладную программу внешнего уровня, если правильным образом настроить стек. Как видно из рис. 72.8, команда ret возврата на внешний уровень (при отсутствии параметров) ожидает наличия в стеке четырех двойных слов: полного указателя адреса перехода и кадра внешнего стека. Если сформировать в стеке такую структуру и выполнить команду дальнего возврата ret, управление будет передано программе внешнего уровня. Чтобы разобраться в деталях такой передачи управления, рассмотрим более подробно механизмы защиты по привилегиям.

Как уже отмечалось, каждому сегменту приписывается некоторый уровень привилегий, определяемый полем DPL в его дескрипторе. Аналогичное поле имеется в любом селекторе, где оно называется RPL, и определяет запрашиваемый уровень привилегий, т. е. уровень привилегий запросчика (незапрашиваемого сегмента!). Указание в поле RPL значения, превышающего значение CPL, позволяет как бы уменьшить привилегии выполняемой программы и тем самым запретить ей обращение к сегменту, использование которого в противном случае было бы разрешено. Такое изменение привилегий источника запроса практикуется при передаче селекторов в качестве параметров в процедуры внешних уровней. Обычно же в поле RPL селектора указывают либо значение CPL текущей программы, либо 0. В этом случае поле RPL не влияет на механизм защиты по привилегиям.

Если программа делает попытку обращения к некоторому сегменту данных (загружая в один из сегментных регистров соответствующий этому сегменту селектор), процессор сравнивает текущий уровень привилегий программы с уровнем привилегий адресуемого сегмента. Программе разрешается доступ к сегменту лишь того же или более внешнего уровня привилегий. Другими словами, при обращении к данным должно выполняться условие CPL<=DPL.

Команды дальних вызовов и переходов call и jmp (как прямые, так и косвенные), осуществляя передачу управления в другой сегмент, изменяют значение селектора, хранящегося в CS. При этом в принципе возникает возможность смены текущего уровня привилегий. Процессор следит за этим действием и разрешает загрузку нового селектора только в том случае, если значение DPL дескриптора, связанного с этим селектором, точно равно значению CPL. Таким образом, непосредственные переходы и вызовы подпрограмм разрешаются только из того же кольца, в котором находится запрашивающая программа.

Очевидно, что это ограничение носит слишком жесткий характер, так как оно делает невозможным обращение прикладных программ к сервису операционной системы. Для смягчения этого ограничения введена возможность вызова процедур из внутренних колец защиты через шлюзы вызова. Поскольку сами шлюзы обычно располагаются в сегментах нулевого кольца, они формируются операционной системой или другой управляющей программой и недоступны программам пользователя. Таким образом, доступ к средствам операционной системы жестко контролируется: прикладная программа может осуществлять вызовы только заранее обусловленных процедур внутренних колец. Обращение к процедурам из внешних колец запрещено безусловно.

Рассмотрим теперь особенности передачи управления процедуре внешнего кольца с помощью команды ret.

На этапе подготовки этого перехода в стек загружается селектор сегмента команд прикладной программы. В процессе выполнения команды ret этот селектор будет загружен в регистр CS, в результате чего его поле RPL определит текущий уровень привилегий CPL. Прикладная программа расположена на уровне 3 и должна работать с текущим уровнем привилегий CPL, равным трем. Поэтому поле RPL в загружаемом селекторе тоже должно иметь значение 3. Дескриптор сегмента команд находится в таблице локальных дескрипторов (бит селектора TI=1) и расположен на втором месте (индекс дескриптора в селекторе равен единице); в результате численное значение селектора оказывается равным 15 (1*8+4+3).

При определении значения селектора стека прикладной программы надо учитывать, что, поскольку для каждого уровня в системе предусмотрен свой стек, процессор при загрузке регистра SS в процедуре перехода с уровня на уровень выполняет такую же жесткую проверку привилегий, как и при вызове процедур. Поэтому для сегмента стека должно выполняться правило RPL=DPL=CPL. Это даст значение селектора сегмента стека 23 (2*8+4+3).

Загрузив в стек нулевого уровня три двойных слова (SS, ESP и EIP), можно выполнить команду ret дальнего возврата из процедуры в режиме 32-битовых операндов (префикс 66h). Так как процедура main, в которую входит эта команда, объявлена по умолчанию ближней (в операторе proc отсутствует описатель far), необходимо указание в явной форме диапазона действия команды (мнемоника retf).

Между прочим, для первого перехода на прикладную программу можно было воспользоваться и командой iret. Для нее, разумеется, требуется несколько иное заполнение стека (рис. 72.9).

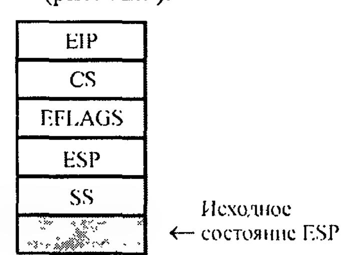


Рис. 72.9. Заполнение стека для перехода на внешний уровень командой iret

Помимо подготовки стека, для правильного выполнения команды iret необходимо позаботиться о сбросе бита NT в регистре флагов. Выше отмечалось, что при загрузке компьютера бит NT может быть установлен. В этом случае команда iret (без

предшествующего аппаратного прерывания) попытается выполнить обратный межзадачный возврат через TSS, что в нашем случае приведет к аварии.

Соответствующий фрагмент программы будет выглядеть так:

```
;Сбросим бит NT для команды iret перехода на прикладную программу
        pushf               ;Перешлем содержимое регистра
        pop     AX          ;флагов через стек в AX
        and     AX,0BFFFh   ;Сбросим бит 14 (4000h)
        push    AX          ;И через стек вернем
        popf                ;в регистр флагов
;Подготовим стек для команды ret перехода на прикладную программу
        mov     EAX,23      ;Селектор сегмента стека
        push    EAX         ;прикладной программы в LDT
        mov     EAX,256     ;Указатель стека
        push    EAX         ;прикладной программы
        pushfd              ;EFLAGS в стек
        mov     EAX,15      ;Селектор сегмента команд
        push    EAX         ;прикладной программы
        mov     AX,offset task ;Входное значение EIP
        push    EAX         ;прикладной программы
;Вызовем прикладную программу
        db      66h         ;"Возврат из подпрограммы"
        iret                ;в прикладную программу
```

Статья 73. **Страничное преобразование**

Как уже отмечалось, адрес, вычисляемый процессором на основе селектора и смещения, относится к линейному адресному пространству, не обязательно совпадающему с физическим. Преобразование линейных адресов в физические осуществляется с помощью так называемой страничной трансляции, частично реализуемой процессором, а частично – операционной системой. Если страничная трансляция выключена, все линейные адреса в точности совпадают с физическими; если страничная трансляция включена, то линейные адреса преобразуются в физические в соответствии с содержимым страничных таблиц (рис. 73.1).

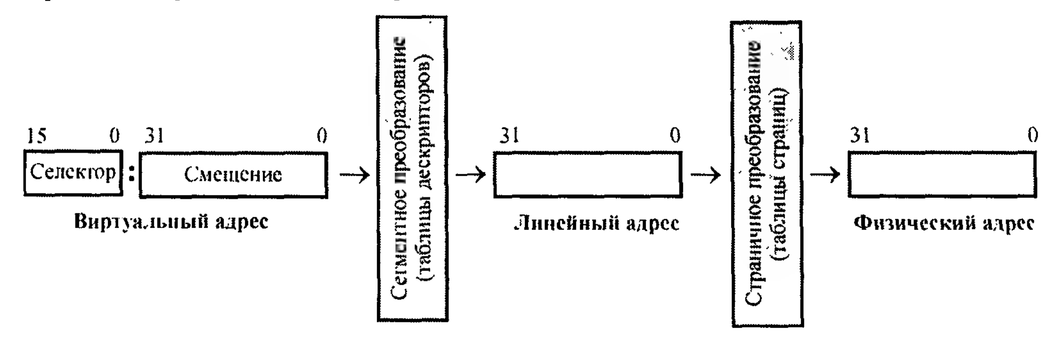


Рис. 73.1. Цепочка преобразований виртуального адреса в физический

Страницей называется связный участок линейного или физического адресного пространства объемом 4 Кбайт. Программа работает в линейном адресном пространстве, не подозревая о существовании страничного преобразования или даже самих страниц. Механизм страничной трансляции отображает линейные страницы на физические в соответствии с информацией, содержащейся в страничных таблицах. В ре-

зультате отдельные 4-килобайтовые участки программы могут реально находиться в любых несвязных друг с другом 4-килобайтовых областях физической памяти. Порядок размещения физических страниц в памяти может не соответствовать (и обычно не соответствует) порядку следования линейных страниц. Более того, некоторые линейные страницы могут перекрываться, фактически сосуществуя в одной и той же области физической памяти. На рис. 73.2 показан возможный вариант отображения части линейного адресного пространства на физическое, полученный в конкретном сеансе работы Windows 95.

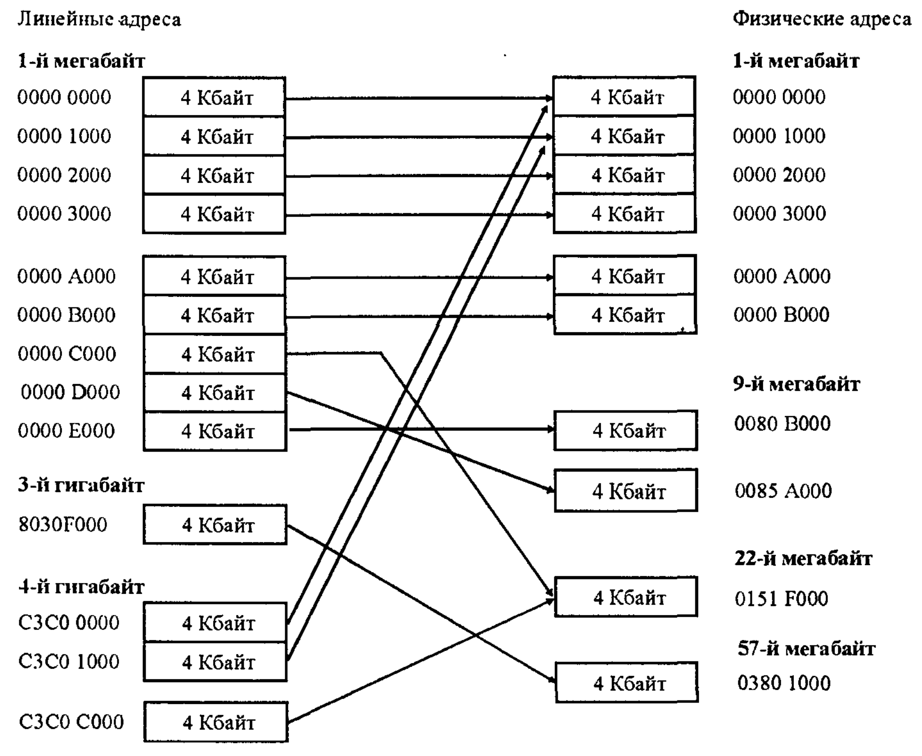


Рис. 73.2. Отображение линейных адресов на физические

Из рисунка видно, что первым 48 Кбайт линейного адресного пространства, зарезервированным для работы MS-DOS, соответствуют те же самые физические адреса. Однако 49-й килобайт (линейный адрес 000C000h) отображается уже на 22-й мегабайт физических адресов, после чего, можно сказать, начинается полная путаница. Приложение Windows, выполняемое в 3-м гигабайте линейного адресного пространства (базовый линейный адрес 8030F000), физически расположено в самом конце памяти, в 57-м мегабайте (компьютер, на котором проводились эти исследования, был укомплектован расширенной памятью общим объемом 64 Мбайт). Наконец, часть адресов 4-го гигабайта повторяют отображение начала линейного адресного пространства (так называемые старшие линейные адреса DOS).

Страничная трансляция представляет собой довольно сложный механизм, в котором принимают участие аппаратные средства процессора и находящиеся в памяти таб-

лицы преобразования адресов (их иногда называют таблицами трансляции). Назначение и взаимодействие элементов системы страничной трансляции схематически изображено на рис. 73.3.

Система страничных таблиц состоит из двух уровней. На первом уровне находится каталог таблиц страниц (или просто каталог страниц) – резидентная в памяти таблица, содержащая 1024 4-байтовых поля с адресами таблиц страниц. На втором уровне находятся таблицы страниц, каждая из которых содержит так же 1024 4-байтовых поля с адресами физических страниц памяти. Максимально возможное число таблиц страниц определяется числом полей в каталоге и может доходить до 1024. Поскольку размер страницы составляет 4 Кбайт, 1024 таблицы по 1024 страницы перекрывают все адресное пространство (4 Гбайт).

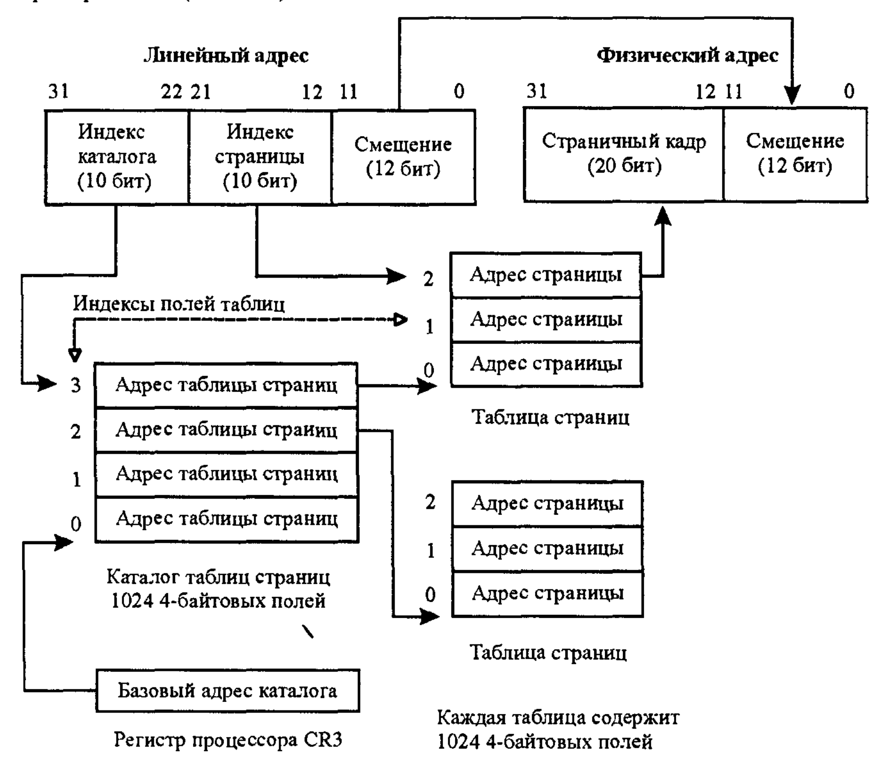


Рис. 73.3. Страничная трансляция адресов

Не все 1024 таблицы страниц должны обязательно иметься в наличии (кстати, они заняли бы в памяти довольно много места – 4 Мбайт). Если программа реально использует лишь часть возможного линейного адресного пространства, а так всегда и бывает, то неиспользуемые поля в каталоге страниц помечаются как отсутствующие. Для таких полей система, экономя память, не выделяет страничные таблицы. С другой стороны, каталог страниц (или страница каталога, как его иногда называют, поскольку весь он имеет размер 4 Кбайт) должен всегда находиться в физической памяти.

При включенной страничной трансляции линейный адрес рассматривается как совокупность трех полей: 10-битового индекса в каталоге страниц, 10-битового индек-

са в выбранной таблице страниц и 12-битового смещения в выбранной странице. Напомним, что линейный адрес образуется путем сложения базового адреса сегмента, взятого из дескриптора сегмента, и смещения в этом сегменте, предоставленного программой.

Старшие 10 бит линейного адреса образуют номер элемента в каталоге страниц. Базовый физический адрес каталога хранится в одном из управляющих регистров процессора, конкретно – в регистре CR3 (см. рис. 64.3). Из-за того, что каталог сам представляет собой страницу и выровнен в памяти на границу 4 Кбайт, в регистре CR3 для адресации к каталогу используются лишь старшие 20 бит, а младшие 12 бит зарезервированы для будущих применений.

Элементы каталога имеют размер 4 байта, поэтому индекс, извлеченный из линейного адреса, сдвигается влево на 2 бита (т. е. умножается на 4) и полученная величина складывается с базовым адресом каталога, образуя адрес конкретного элемента каталога. Каждый элемент каталога содержит физический базовый адрес одной из таблиц страниц, причем, поскольку таблицы страниц сами представляют собой страницы и выровнены в памяти на границу 4 Кбайт, в этом адресе значащими являются только старшие 20 бит.

Далее из линейного адреса извлекается средняя часть (биты 12...21), сдвигается влево на 2 бита и складывается с базовым адресом, хранящимся в выбранном поле каталога. В результате образуется физический адрес страницы в памяти, в котором опять же используются только старшие 20 бит. Этот адрес, рассматриваемый как старшие 20 бит физического адреса адресуемой ячейки, носит название страничного кадра. Страничный кадр дополняется с правой стороны младшими 12 битами линейного адреса, которые проходят через страничный механизм без изменения и играют роль смещения внутри выбранной физической страницы.

Для того чтобы работать с таблицами страничной трансляции, необходимо иметь представление о содержащейся в них информации. И каталог таблиц страниц, и сами таблицы страниц имеют одинаковый формат элементов, показанный на рис. 73.4.

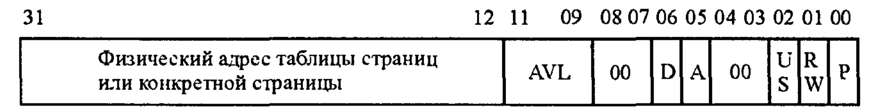


Рис. 73.4. Формат элемента таблиц страниц и таблицы каталога

Бит присутствия P (Present) указывает, содержит ли данный элемент значащую информацию. Если P=0, процессор не интерпретирует остальные биты элемента. Обращение в процессе страничного преобразования к элементу с P=0 вызывает исключение 14 (страничное нарушение), которое может заставить операционную систему загрузить в память с диска отсутствующую в данный момент таблицу страниц.

Бит R/W (Read/Write) характеризует права выполнения. Если этот бит равен единице, страница может читаться, записываться или выполняться. Если бит равен нулю, страницу можно читать и записывать, но не выполнять. Бит R/W игнорируется процессором на уровнях 0, 1 и 2. Этот бит в каталоге страниц относится ко всем страницам, отображаемым через данный элемент.

Бит U/S (User/Supervisor) характеризует права доступа. Если этот бит равен единице, страница доступна программам, выполняющимся на любом уровне привилегий,

включая уровень 3. Если бит равен нулю, страница доступна только для программ уровней 0, 1 или 2. Этот бит в каталоге страниц относится ко всем страницам, отображаемым через данный элемент.

Бит А (Accessed) является индикатором обращения. Бит А в элементе таблицы страниц устанавливается процессором в 1 перед любым обращением к любой странице, отображаемой через данный элемент. Процессор никогда не сбрасывает бит А; этот бит может периодически сбрасываться операционной системой для получения информации о реальном использовании страниц.

Бит мусора D (Dirty) в таблицах страниц устанавливается процессором в 1 перед любым обращением с целью записи к отображаемой через данный элемент странице. Процессор не модифицирует бит D элементов каталога.

Поле AVL не используется процессором и предназначено для произвольного использования программным обеспечением.

Рассмотрим пример (полученный при выполнении конкретного приложения Windows), позволяющий проследить цепочку преобразования виртуального адреса в физический. Пусть программа выполняет команду

```
mov    DS:[BX],0010h
```

при этом содержимое DS (селектор) составляет 29D7h, а содержимое BX (смещение) – 004Fh. Кстати, из значения селектора видно, что описываемому им сегменту присвоен низший уровень привилегий (DPL=3), а дескриптор сегмента входит в таблицу локальных дескрипторов (TI=1).

Старшие 13 бит селектора образуют индекс дескриптора в системной дескрипторной таблице. Как уже говорилось выше, каждый дескриптор включает в себя довольно большой объем информации о конкретном сегменте и, в частности, его линейный адрес. Пусть в ячейке дескрипторной таблицы со смещением 29D0h записан линейный базовый адрес сегмента 80295CC0h. Заметим, что этот адрес принадлежит 3-му гигабайту линейного адресного пространства. Тогда полный линейный адрес адресуемой ячейки определится как сумма базового адреса и смещения:

```
Базовый адрес сегмента  80295CC0h
Смещение                0000004Fh
Полный линейный адрес   80295D0Fh
```

При выключенной табличной трансляции величина 80295D0Fh представляла бы собой абсолютный физический адрес ячейки, в которую приведенная выше команда mov должна была записать число 0010h. Однако страничная адресация в современных операционных системах всегда включена и линейный адрес отнюдь не совпадает с физическим.

Посмотрим, как будет образовываться физический адрес при использовании страничной трансляции адресов. Полученный линейный адрес надо разделить на три составляющие для выделения индексов и смещения (рис. 73.5)

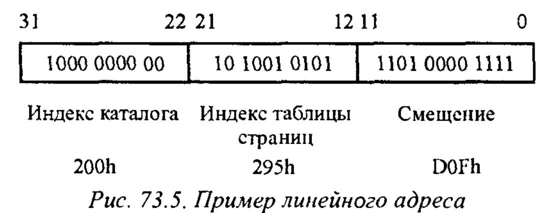


Рис. 73.5. Пример линейного адреса

Индекс каталога составляет 200h. Умножение его на 4 даст смещение от начала каталога. Это смещение равно 800h.

Индекс таблицы страниц оказался равным 295h. После умножения на 4 получаем смещение в таблице страниц, равное в данном случае A54h.

Предположим, что регистр CR3 содержит число 00820000h. Тогда физический адрес ячейки в каталоге, откуда надо получить адрес закрепленной за данным участком программы таблицы страниц, составит 00820000h + 800h = 00820800h. Пусть по этому адресу записано число 00861267h. Его 12 младших бит (число 267h) составляют служебную информацию (в частности, бит 1 свидетельствует о присутствии этой таблицы страниц в памяти, а бит 5 говорит о том, что к этой таблице уже были обращения), и их не следует рассматривать как часть адреса. Записав вместо этих битов нули, получим число 00861000h, которое является физическим базовым адресом таблицы страниц. Для получения адреса требуемой ячейки этой таблицы к базовому адресу надо прибавить смещение A54h. Результирующий адрес составит 00861A54h.

Будем считать, что по адресу 00861A54h записано число 015BA267h. Опять, отбросив служебные биты, получим адрес физической страницы в памяти 015BA000h. Этот адрес всегда оканчивается тремя нулями, так как страницы выровнены в памяти на границу 4 Кбайт. Для получения физического адреса адресуемой ячейки следует заполнить 12 младших бит полученного адреса битами смещения из линейного адреса нашей ячейки, в которых в нашем примере записано число D0Fh. В итоге получаем физический адрес памяти 015BACD6h, расположенный в конце 22-го мегабайта.

Как видно из этого примера, и со страничной трансляцией и без нее вычисление физических адресов адресуемых ячеек выполняется в защищенном режиме совсем не так, как в реальном. Неприятным практическим следствием правил адресации защищенного режима является уже упоминавшаяся "оторванность" прикладной программы от физической памяти. Программист, отлаживающий программу защищенного режима (например, приложение Windows), может легко заглянуть в сегментные регистры и определить селекторы, выделенные программе. Однако селекторы абсолютно ничего не говорят о физических адресах, используемых программой. Даже если ухитриться заглянуть в таблицу дескрипторов (это можно сделать только с помощью специального системного отладчика), то и тогда мы узнаем лишь линейный, а не физический адрес. Для получения физического адреса следует изучить содержимое таблиц трансляции. Поскольку и таблицы дескрипторов и таблицы трансляции недоступны прикладной программе, программист не знает, где в памяти находится его программа или используемые ею области данных.

С другой стороны, использование в процессе преобразования адресов защищенных системой таблиц имеет свои преимущества. Обычно многозадачная операционная система создает для каждой выполняемой задачи свой набор таблиц преобразования адресов. Это позволяет каждой из задач использовать весь диапазон виртуальных адресов, при этом, хотя для разных задач виртуальные адреса могут совпадать (и как правило, по крайней мере частично совпадают), сегментное и страничное преобразования обеспечивают выделение для каждой задачи несовпадающих областей физической памяти, надежно изолируя виртуальные адресные пространства задач друг от друга.

Раздел седьмой

ПРИКЛАДНЫЕ ВИРТУАЛЬНЫЕ ДРАЙВЕРЫ СИСТЕМ WINDOWS 95/98

Статья 74. **Виртуальные драйверы и виртуальные машины Windows**

Работа операционных систем Windows 95/98 (как и Windows 3.1) в значительной степени основана на использовании специального рода программ – так называемых виртуальных драйверов устройств (или просто виртуальных драйверов, virtual device). Основное назначение виртуального драйвера – виртуализация устройства, т. е. возможность нескольким приложениям одновременно использовать одно и то же физическое устройство. Например, виртуальный драйвер дисплея (VDD) обеспечивает многооконный режим, в котором каждое приложение, выводя информацию на экран, считает, что весь физический экран находится в его распоряжении, в то время как в действительности вывод приложения поступает в выделенное для него окно. Виртуальный контроллер прерываний (VPICD) дает возможность нескольким приложениям совместно использовать единую систему прерываний компьютера. Виртуальный драйвер клавиатуры (VKD) позволяет вводить с клавиатуры символьные строки в любую из выполняемых программ. Разработка нового виртуального драйвера может понадобиться при установке на компьютер новой аппаратуры (или нового программного обеспечения, предназначенного для обслуживания других приложений), которая будет использоваться в многозадачном режиме и для которой в системе Windows не предусмотрено средств виртуализации.

Другое, возможно, более важное для прикладного программиста приложение виртуальных драйверов состоит в их использовании в качестве универсального инструментального средства. Дело в том, что виртуальный драйвер работает в плоской модели памяти на нулевом уровне привилегий. Плоской моделью памяти называется такая организация адресного пространства, когда в дескрипторе указаны нулевой базовый линейный адрес сегмента и предел, соответствующий максимальному размеру сегмента – 4 Гбайт. Такой дескриптор может входить как в локальную, так и в глобальную таблицу дескрипторов. В первом случае речь идет о приложениях, работающих в плоской модели памяти на уровне привилегий 3 (это характерно для 32-разрядных приложений Windows); во втором – о системных компонентах, в частности о виртуальных драйверах. Для виртуального драйвера доступно все линейное адресное пространство (4 Гбайт), все программные составляющие Windows (в том числе, системные виртуальные драйверы) и все программно-управляемые аппаратные средства процессора и компьютера в целом. Виртуальный драйвер представляет собой идеальную среду для исследования системы Windows и контролируемого вмешательства в ее работу. По-

этому решение каких-либо нестандартных задач, например разработка программ управления измерительным оборудованием, подключенным к компьютеру, может потребовать написания специфических виртуальных драйверов, смысл использования которых состоит совсем не в виртуализации аппаратно-программных средств, а в возможности выполнения действий, допустимых лишь на нулевом уровне привилегий (например, обращения к запрещенным портам или к системе прерываний).

Следует заметить, что широкое использование персональных компьютеров для управления экспериментальными установками и производственными процессами определило возрастающий интерес прикладных программистов к проблемам разработки драйверов для систем Windows. В то же время получить сведения по этим вопросам можно практически лишь из справочников, входящих в состав пакетов DDK (Device Development Kit, пакет для разработки драйверов). Эти справочники отличаются строгостью и полнотой, однако совершенно не приспособлены для начального ознакомления с предметом. Мы надеемся, что последние разделы настоящей книги в какой-то степени восполнят этот пробел.

Разработка прикладных виртуальных драйверов требует основательного знакомства с особенностями работы современных процессоров в защищенном режиме, а также с некоторыми внутренними алгоритмами функционирования системы Windows. Защищенный режим был рассмотрен в предыдущем разделе; настоящая статья посвящена краткому обзору тех специальных понятий и средств Windows, знание которых необходимо для написания программ виртуальных драйверов.

Одним из базовых понятий системы Windows является понятие виртуальной машины. Согласно документации Microsoft, виртуальной машиной (VM) называется выполняемая задача, состоящая из приложения, поддерживающего программного обеспечения (например, программ DOS и BIOS), памяти и регистров процессора. Каждая виртуальная машина обладает собственным адресным пространством, пространством ввода-вывода и таблицей векторов прерываний. В адресное пространство VM отображаются ПЗУ BIOS, драйверы и другие программы DOS, а также загруженные до запуска Windows резидентные программы, что обеспечивает доступ к ним прикладных программ, выполняемых под управлением системы Windows. В состав виртуальной машины входят виртуальные аппаратные регистры, в частности виртуальные маски прерываний, виртуальные флаги процессора и др.

Первая виртуальная машина, создаваемая после загрузки Windows и называемая системной виртуальной машиной, в системах Windows 95/98 предназначена для выполнения 16- и 32-разрядных приложений Windows. Для каждого сеанса DOS создается своя виртуальная машина, и таким образом одновременно в системе может существовать несколько виртуальных машин.

Управление виртуальными машинами возлагается на менеджер виртуальных машин (Virtual machine manager, VMM), являющийся ядром операционных систем Windows 95/98. VMM представляет собой 32-разрядную систему защищенного режима, работающую на нулевом уровне привилегий в плоской модели памяти. В функции VMM входит создание, управление и завершение виртуальных машин, а также управление памятью, процессами, прерываниями и нарушениями защиты. Так, если приложение делает попытку записи или чтения по адресам памяти, не принадлежащим данной виртуальной машине, или выполняет команды ввода-вывода в запрещенные порты, процессор возбуждает исключение и управление передается VMM, который

анализирует причину исключения и либо с помощью виртуальных драйверов обеспечивает выполнение затребованной операции, либо аварийно завершает приложение.

Для обеспечения функционирования VMM и виртуальных драйверов система создает два глобальных селектора: 28h для программных кодов и 30h для системных полей данных. Дескрипторы обоих селекторов почти одинаковы: в них указано нулевое значение DPL, нулевой базовый линейный адрес и граница 4 Гбайт (рис. 74.1). В регистр CS загружается селектор 28h, во все остальные сегментные регистры (DS, SS, ES, FS и GS) – селектор 30h. Ни VMM, ни виртуальные драйверы никогда не изменяют содержимое сегментных регистров.

```
:GDT 30
Sel    Type      Base      Limit     DPL   Attributes
0030   Data32    00000000  FFFFFFFF  0     P   RW
```

Рис. 74.1. Дескриптор 30h (вывод отладчика SoftICE, см. статью 77)

VMM вместе с системными виртуальными драйверами предоставляет большое количество служебных функций, позволяющих виртуальным драйверам активизировать по ходу своего выполнения те или иные системные процедуры и алгоритмы. Например, функция VMM _MapPhysToLinear отображает заданный участок физических адресов на линейное адресное пространство, что дает возможность виртуальному драйверу выполнять чтение или запись в физической памяти; функции Install_Handler и Install_Mult_IO_Handlers устанавливают в системе прикладные процедуры обслуживания заданных портов; функция Simulate_Push помещает указанный параметр в стек приложения. Служебные функции имеются и у системных виртуальных драйверов. Так, виртуальный контроллер прерываний VPICD позволяет с помощью функции VPICD_Virtualize_IRQ организовать прикладную обработку аппаратных прерываний, а виртуальный таймер VTD предоставляет функции VTD_Begin_Min_Period и VTD_End_Min_Period для управления частотой системного таймера. Служебные функции VMM и системных виртуальных драйверов, в отличие от функций DOS или Windows, не могут вызываться непосредственно из приложения; обращение к ним возможно только из виртуальных драйверов.

Взаимодействуя с VMM и виртуальными машинами, виртуальные драйверы используют целый ряд системных полей и структур данных. Одним из важнейших системных идентификаторов, создаваемых VMM для каждой виртуальной машины, является ее дескриптор. Дескриптор однозначно идентифицирует виртуальную машину и используется при вызовах функций VMM для задания VM, к которой обращено требуемое действие. Заметим, что дескриптор виртуальной машины является базовым адресом, от которого ведется отсчет адресов ряда важных полей VMM. Так, в двойном слове по адресу [дескриптор VM – 8] располагается адрес таблицы векторов прерываний защищенного режима.

Дескриптор VM фактически является линейным 32-разрядным адресом управляющего блока данной VM. Управляющий блок представляет собой структуру из пяти двойных слов, содержащую информацию о виртуальной машине. Эта структура определена в файле VMM.INC и имеет следующий вид:

```
cb_s    struc
CB_VM_Status        dd   ?  ; Слово состояния VM
CB_High_Linear      dd   ?  ; Старший линейный адрес
CB_Client_Pointer   dd   ?  ; Адрес структуры клиента
CB_VMID             dd   ?  ; Идентификатор VM
```

```
CB_Signature        dd   ?  ; Сигнатура
cb_s    ends
```

В слове состояния VM каждый бит закреплен за определенной характеристикой текущего состояния VM. Так, установленный бит 1 (символическое обозначение VMStat_Background) говорит о том, что VM выполняется в фоновом режиме; бит 6 (VMStat_PM_App) свидетельствует о наличии в VM приложения защищенного режима, бит 7 (VMStat_PM_Use32) характеризует 32-разрядное приложение защищенного режима и т. д. Иногда в процессе отладки виртуального драйвера приходится обращаться к этому слову для определения состояния виртуальной машины.

Старший линейный адрес заслуживает особого рассмотрения. При запуске в рамках Windows одного или нескольких сеансов DOS (и создания соответствующих виртуальных машин) происходит копирование 1-го мегабайта физического адресного пространства в различные участки расширенной памяти. В каждой копии DOS имеется своя таблица векторов прерываний, своя видеопамять и даже своя копия ПЗУ BIOS, что дает возможность приложениям DOS различных сеансов параллельно и независимо работать каждому со своей памятью, не разрушая память других сеансов. Для того чтобы с сеансами DOS могли взаимодействовать VMM или виртуальные драйверы, физическая память сеансов DOS отображается на некоторые линейные адреса (4-го гигабайта линейного адресного пространства), которые и называются старшими линейными адресами. Для каждой VM существует собственный старший адрес (рис. 74.2).

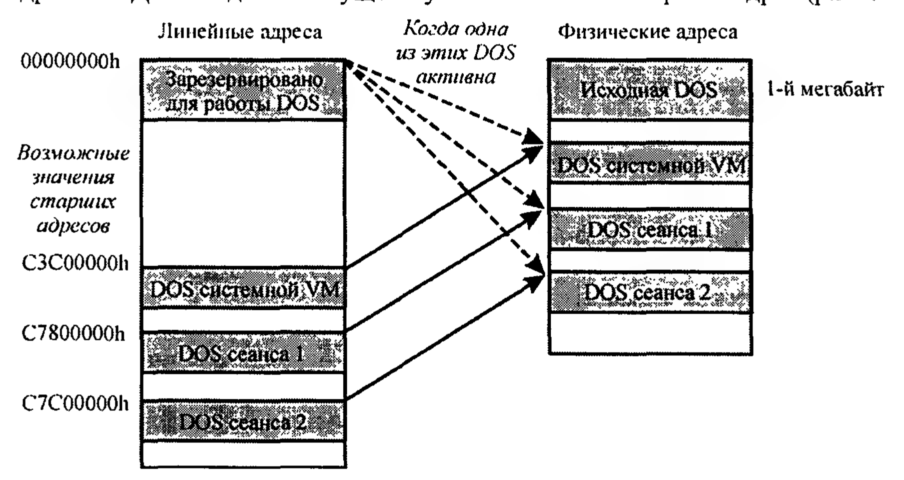


Рис. 74.2. Линейные и физические адреса сеансов DOS

Сами программы DOS могут работать только в 1-м мегабайте линейных адресов (вспомним, что в реальном режиме физические адреса совпадают с линейными, а DOS располагается в 1-м мегабайте физических адресов). Поэтому для активного в настоящий момент сеанса DOS его физические адреса отображаются еще и на 1-й мегабайт линейных. В результате любое активное приложение DOS, работающее в пределах 1-го мегабайта линейных адресов, обращается только к собственной копии DOS, входящей в соответствующую виртуальную машину. В то же время отображение на старшие линейные адреса действует всегда, что позволяет виртуальным драйверам обращаться к физическому адресному пространству любой виртуальной машины; более того, виртуальные драйверы имеют право взаимодействовать с сеансами DOS

исключительно через старшие линейные адреса. При этом линейный адрес любой адресуемой ячейки определяется суммированием ее физического адреса со старшим линейным адресом соответствующей VM.

Структура клиента, адрес которой CB_Client_Pointer входит в управляющий блок, является важнейшим набором данных, обеспечивающим обмен информацией между приложением и виртуальным драйвером. Когда при вызове виртуального драйвера процессор переходит на нулевой уровень привилегий, в стеке уровня 0 (не в стеке приложения!) сохраняются значения всех регистров вызывающего приложения, соответствующие точке вызова драйвера. Совокупность этих значений и называется структурой клиента. При входе в драйвер адрес структуры клиента находится в регистре EBP, а обращение к отдельным элементам этой структуры осуществляется с помощью наглядных символических обозначений вида Client_AX, Client_FLAGS, Client_CS и т. д.:

```
movzx  EAX,[EBP.Client_AX]    ;Передача AX приложения в EAX драйвера
mov    [EBP.Client_DX],data   ;Передача данного из драйвера
                              ;в DX приложения
```

Возможно также обращение как к байтовым регистрам (Client_AL, Client_DH), так и к 32-разрядным (Client_EFLAGS, Client_ESI). Структура клиента является чрезвычайно удобным средством обмена информацией между приложением и драйвером, так как регистры приложения не только сохраняются в структуре клиента при переходе в драйвер, но и восстанавливаются из нее при возврате в приложение. Поэтому изменение значений в структуре клиента в процессе выполнения программы драйвера приводит к соответствующему изменению регистров процессора после возврата в приложение.

Другие аспекты взаимодействия системы Windows, приложений и виртуальных драйверов будут рассмотрены в последующих статьях.

Статья 75. **Структура виртуального драйвера**

Виртуальные драйверы обычно пишутся на языке ассемблера, хотя внутренние процедуры драйвера вполне могут составляться на языке Си (Си++). Использование языка высокого уровня упрощает реализацию вычислительных и логических алгоритмов в процедурах драйвера, но, с другой стороны, делает текст драйвера менее наглядным. Поскольку нашей задачей является освоение основных принципов разработки драйверов и приводимые ниже примеры относительно просты, мы ограничимся языком ассемблера. Для создания выполнимого модуля надо использовать 32-разрядные ассемблер и компоновщик, в частности входящие в пакет DDK. В том же пакете можно найти заголовочные и другие включаемые файлы с многочисленными макросами и определениями констант, используемые при разработке драйверов.

Рассмотрим прежде всего общую структуру виртуального драйвера, а также правила его подготовки и загрузки. В примере 75.1 приведен текст простейшего виртуального драйвера для систем Windows 95/98, который не выполняет никакой работы, хотя и может быть установлен в системе.

Пример 75.1. Структура виртуального драйвера

```
.386p                      ;Будут использоваться 32-разрядные регистры
.XLIST                     ;Подавление включения в листинг текста vmm.inc
include vmm.inc            ;Подключение файла vmm.inc
.LIST                      ;Разрешение дальнейшего вывода в листинг
;Блок описания драйвера
```

```
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h, \
Undefined_Init_Order,V86_API_Handler,PM_API_Handler
;==============Сегмент инициализации реального режима==============
VxD_REAL_INIT_SEG
;Процедура инициализации реального режима
BeginProc VMyD_Real_Init   ;Точка входа при инициализации
        mov   AH,09h       ;Функция DOS вывода на экран
        mov   DX,offset msg;Адрес выводимого сообщения
        int   21h          ;Вызов DOS
        mov   AX,Device_Load_Ok;Код успешного завершения
        xor   BX,BX        ;Дополнительные
        xor   SI,SI        ;данные
        xor   EDX,EDX      ;отсутствуют
        ret
msg db 'Виртуальный драйвер VMyD загружен',13,10,'$'
EndProc   VMyD_Real_Init
VxD_REAL_INIT_ENDS
;===============Сегмент данных=========================================
VxD_DATA_SEG
;Здесь размещаются данные, используемые драйвером
VxD_DATA_ENDS
;===============Сегмент защищенного режима============================
VxD_CODE_SEG
;Управляющая процедура для обработки системных сообщений Windows
BeginProc VMyD_Control
;Сюда могут быть включены процедуры обработки различных
;системных сообщений Windows: Device_Init инициализации драйвера,
;Create_VM создания новой виртуальной машины, Begin_PM_App запуска
;приложения защищенного режима и End_PM_App его завершения и др.
        clc
        ret
EndProc VMyD_Control
;Процедура, вызываемая из приложения реального режима (приложения DOS)
BeginProc V86_API_Handler
;Сюда включаются содержательные процедуры драйвера, которые
;должны выполняться при вызове драйвера из приложений реального режима
        ret                ;Возврат из драйвера в вызывающее приложение
EndProc V86_API_Handler
;Процедура, вызываемая из приложения защищенного режима (приложения Windows)
BeginProc PM_API_Handler
;Сюда включаются содержательные процедуры драйвера, которые
;должны выполняться при вызове драйвера из приложений защищенного режима
        ret                ;Возврат из драйвера в вызывающее приложение
EndProc PM_API_Handler
VxD_CODE_ENDS
end VMyD_Real_Init         ;Конец текста драйвера с указанием точки входа
```

Исходный текст драйвера начинается с директивы ассемблера .386р, позволяющей использовать расширенный набор команд современных процессоров и, в частности, привилегированных команд защищенного режима. Следующая далее директива .XLIST подавляет вывод в листинг трансляции всех последующих предложений программы вплоть до отмены ее действия директивой .LIST. В нашем случае подавляется вывод в листинг весьма обширного файла VMM.INC с определениями макросов и констант, фактически обеспечивающего использование в драйвере средств VMM.

Как видно из приведенного листинга, драйвер в простейшем случае состоит из блока описания драйвера и трех сегментов: сегмента инициализации реального режима VxD_REAL_INIT, в котором размещается процедура инициализации реального режима (вместе с относящимися к ней данными); сегмента данных VxD_DATA_SEG со всеми дан-

ными, которые могут потребоваться драйверу в процессе его работы; сегмента команд защищенного режима VxD_CODE, содержащего в нашем случае процедуры VMyD_Control, V86_API_Handler и PM_API_Handler (имена процедур произвольны).

В тексте драйвера широко используются макросы, определенные в файле VMM.INC. Так, макросы VxD_REAL_INIT_SEG и VxD_REAL_INIT_ENDS задают начало и конец сегмента инициализации реального режима, макросы VxD_CODE_SEG и VxD_CODE_ENDS – начало и конец сегмента защищенного режима, а макросы BeginProc и EndProc – границы процедур, входящих в состав драйвера. Имена макросов можно набирать как строчными, так и прописными буквами. Приведенное в примере начертание взято из документации Windows.

Блок описания драйвера, с которого начинается текст драйвера, вводится с помощью макроса Declare_Virtual_Device. Этот макрос требует указания восьми параметров, располагаемых в следующем порядке:

- имя драйвера;
- номер версии и подверсии;
- имя управляющей процедуры;
- идентификационный номер драйвера;
- порядок инициализации;
- имя процедуры обслуживания реального режима;
- имя процедуры обслуживания защищенного режима.

В нашем случае драйверу дано произвольное имя VMyD, управляющей процедуре (которая обязательно должна присутствовать, хотя у нас она фактически пуста) – VMyD_Control, процедурам обслуживания реального и защищенного режимов, которые в дальнейшем мы будем называть API-процедурами (Application Program Interface, интерфейс прикладных программ) – V86_API_Handler и PM_API_Handler соответственно. Для версии выбран номер 1.0, а для идентификационного кода – 8000h. При разработке коммерческого виртуального драйвера его следует зарегистрировать в корпорации Microsoft и получить для него идентификационный код. Порядок инициализации важен для драйверов, обращающихся друг к другу; для наших драйверов порядок инициализации не имеет значения, что указано с помощью константы Undefined_Init_Order.

Процедура инициализации реального режима VMyD_Real_Init вызывается автоматически в процессе загрузки драйвера, когда процессор еще не перешел в защищенный режим. Поскольку процедура выполняется в реальном режиме, в ней возможен вызов функций DOS. У нас она используется для вывода на экран (в процессе загрузки Windows) сообщения о загрузке нашего драйвера. При входе в процедуру инициализации реального режима регистры CS, DS и ES имеют одно и то же значение, равное сегментному адресу сегмента реального режима, а регистр IP содержит 0. Таким образом, процедура инициализации должна начинаться с программных строк, вслед за которыми могут располагаться любые поля данных, используемые на этом этапе инициализации драйвера.

API-процедуры являются, можно сказать, содержательной частью драйвера. Эти процедуры, выполняемые на нулевом уровне привилегий в плоской модели памяти, могут быть вызваны из любой прикладной программы (выполняемой, естественно, на третьем уровне защиты). Для обращения к драйверу из приложений DOS (реальный режим) и Windows (защищенный режим) предусматриваются две различные API-

процедуры; если действия драйвера в ответ на обращение программ в обоих случаях одинаковы, в соответствующих полях блока описания драйвера можно указать имя единственной процедуры. В некоторых случаях драйвер работает сам по себе, без обращения к нему из программы пользователя, тогда как сами процедуры, так и их имена в блоке описания драйвера отсутствуют. В дальнейшем мы будем рассматривать только драйверы, предназначенные для работы с приложениями Windows и в блоке описания драйвера имя API-процедуры реального режима будет отсутствовать:

```
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h, \
    Undefined_Init_Order,,API_Handler
```

Для ускорения процесса подготовки выполнимого модуля виртуального драйвера удобно создать пакет из трех файлов. Приводимый ниже пример их содержимого предполагает, что пакет DDK установлен на диске G: в каталоге DDK95, исходный файл с текстом драйвера имеет имя VMYD.ASM, а выполнимый модуль драйвера – VMYD.VXD.

Файл WIN95.BAT

```
g:\ddk95\masm611c\ml /coff /DBLD_COFF /Zi /Fl /Sn /DMASM6 /c /Cx vmyd.asm
g:\ddk95\msvc20\link vmyd.obj @vmyd.lnk
del vmyd.obj
```

Файл VMYD.DEF

```
LIBRARY VMYD
DESCRIPTION 'Virtual VMyD driver'
EXETYPE DEV386
SEGMENTS
_LTEXT CLASS 'LCODE' PRELOAD NONDISCARDABLE
_LDATA CLASS 'LCODE' PRELOAD NONDISCARDABLE
_RCODE CLASS 'RCODE'
EXPORTS
VMyD_DDB @1
```

Файл VMYD.LNK

```
DEBUG
DEBUGTYPE:BOTH
MACHINE:I386
PDB:NONE
DEF:VMYD.DEF
OUT:VMYD.VXD
VXD
```

Для того чтобы в операторе include в исходном тексте драйвера не описывать путь к файлу VMM.INC, можно в файл AUTOEXEC.BAT включить команду

```
SET INCLUDE=g:\ddk95\inc32
```

Загружаемый модуль драйвера подготавливается запуском в сеансе DOS описанного выше командного файла W95.BAT.

Установка созданного драйвера в системе осуществляется путем включения в раздел [386Enh] файла SYSTEM.INI следующей строки (в предположении, что программа драйвера находится в каталоге DRIVERS диска F:):

```
DEVICE=f:\drivers\vmyd.vxd
```

В процессе загрузки системы Windows на одном из этапов (после того, как появится и исчезнет полноэкранная заставка Windows) на экран будет выведено сообщение "Виртуальный драйвер VMyD загружен", после чего система будет работать как обыч-

но, поскольку наш драйвер и по собственной инициативе ничего не делает, и не взаимодействует ни с системой, ни с приложениями.

Статья 76. **Взаимодействие драйвера и приложения**

Рассмотрим теперь проблему взаимодействия драйвера с приложением. Эта проблема имеет два аспекта:

- как вызвать из прикладной программы API-процедуру драйвера;
- каким образом передать драйверу исходную информацию и получить результаты его работы.

Следует заметить, что если взаимодействие с драйвером приложений DOS и 16-разрядных приложений Windows осуществляется схожим образом, то для 32-разрядных приложений Windows используются совершенно другие методы. Рассмотрим сначала интерфейс виртуального драйвера с 16-разрядным приложением Windows. Взаимодействию виртуальных драйверов с 32-разрядными приложениями Windows будут посвящены завершающие статьи этого раздела.

Разработаем драйвер, который по запросу приложения будет возвращать в него дескриптор LDT, соответствующий селектору сегмента команд приложения. Из содержимого дескриптора можно будет извлечь информацию о линейном базовом адресе сегмента команд и определить таким образом, в какой области линейного адресного пространства располагается наша прикладная программа. Кроме линейного базового адреса, в дескриптор входит граница сегмента, а также его атрибуты (см. статью 65). Прямое обращение к таблицам дескрипторов позволит нам вспомнить их назначение и состав и послужит полезным упражнением при изучении защищенного режима.

Как уже отмечалось, алгоритм API-процедуры тесно связан с алгоритмом обращения к драйверу из прикладной программы. Действительно, если приложение передает драйверу какие-либо данные, драйвер должен содержать команды приема и сохранения этих данных; если драйвер передает в приложение результаты своей работы, в приложении должны быть строки приема и обработки этой информации. Фактически алгоритм прикладной программы (в части взаимодействия ее с драйвером) определяет состав API-процедуры (или процедур) драйвера. Поэтому мы начнем разработку нашего драйвера с написания взаимодействующей с ним прикладной программы.

Вызывать драйвер можно из приложения, написанного на любом языке, в частности на языке ассемблера. Однако приложения Windows на языке ассемблера оказываются чрезвычайно громоздкими; их составление требует от программиста не только понимания многочисленных внутренних концепций системы Windows, но и освоения целого ряда специфических средств и приемов программирования на языке ассемблера, без которых написать приложение Windows затруднительно. Желающие познакомиться с принципами составления на языке ассемблера прикладных программ для системы Windows отсылаются к 3-му изданию книги К. Г.Финогенова "Основы языка ассемблера" (М.: Радио и связь, 2001).

Гораздо проще написать приложение Windows на языке Си (или Си++). Разумеется, для разработки красивого современного приложения со сложным интерфейсом – меню, диалоговыми окнами, кнопками, курсорами, рисунками и прочим необходимо достаточно свободно владеть всем огромным арсеналом изобразительных средств Windows, однако мы здесь ограничимся относительно простыми программами с ми-

нимом интерфейсных деталей. Краткие пояснения помогут читателю понять суть дела; желающие более основательно изучить программирование для Windows на языке Си++, могут обратиться к книге К. Г.Финогенова "Прикладное программирование для Windows на Borland C++" (Обнинск: Принтер, 1999).

В приводимом ниже примере строки программы пронумерованы, хотя в программах на языке Си++, где допустим переход на следующую строку текста почти в любом месте предложения языка, как и расположение любого числа предложений языка на одной строке текста, эта нумерация носит довольно условный характер.

Приложения Windows готовятся, как правило, с помощью интегрированной среды разработки (Integrated Development Environment, IDE), которая обеспечивает весь цикл разработки приложения (подготовку исходного текста, трансляцию, пробное исполнение и отладку) в полноэкранном интерактивном режиме. Сама IDE запускается из среды Windows и широко использует стандартные средства графического интерфейса Windows – меню, диалоговые окна, инструментальные кнопки и пр.

Подготавливать и отлаживать 16-разрядные приложения Windows проще всего в интегрированной среде Borland C++ версии 4.5. Для работы с 32-разрядными приложениями лучше использовать пакеты Borland C++ 5.0 или Microsoft Visual C++. Поскольку ближайшие несколько статей будут посвящены 16-разрядным приложениям Windows, мы очень коротко рассмотрим основные правила подготовки приложений в среде Borland C++ 4.5.

IDE Borland C++ 4.5 содержит большое количество настраиваемых параметров, доступ к которым осуществляется в основном через команду Options главного меню (рис. 76.1). К счастью, о большинстве этих параметров можно не заботиться, так как их значения по умолчанию устанавливаются разумным образом. Однако кое-что все же придется настроить и проверить.

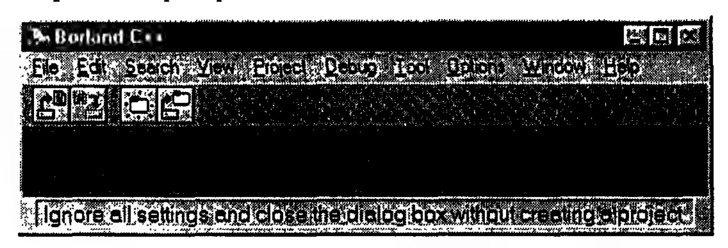


Рис. 76.1. Начальный кадр IDE Borland C++ 4.5

Прежде всего следует установить значения каталогов пользователя, в которых будут храниться исходные (Source), промежуточные (Intermediate) и конечные (Final) файлы, возникающие при разработке программного проекта. Доступ к кадру задания каталогов осуществляется выбором команды Options➤Project➤Directories. На рис. 76.2 для всех файлов программного проекта задан один каталог F:\EXAMPLES.

В том же кадре стоит проверить правильность задания каталогов для включаемых (Include) и библиотечных (Library) файлов. Если IDE установлена стандартным образом, то для этих файлов должны быть указаны каталоги \bc45\include и \bc45\lib.

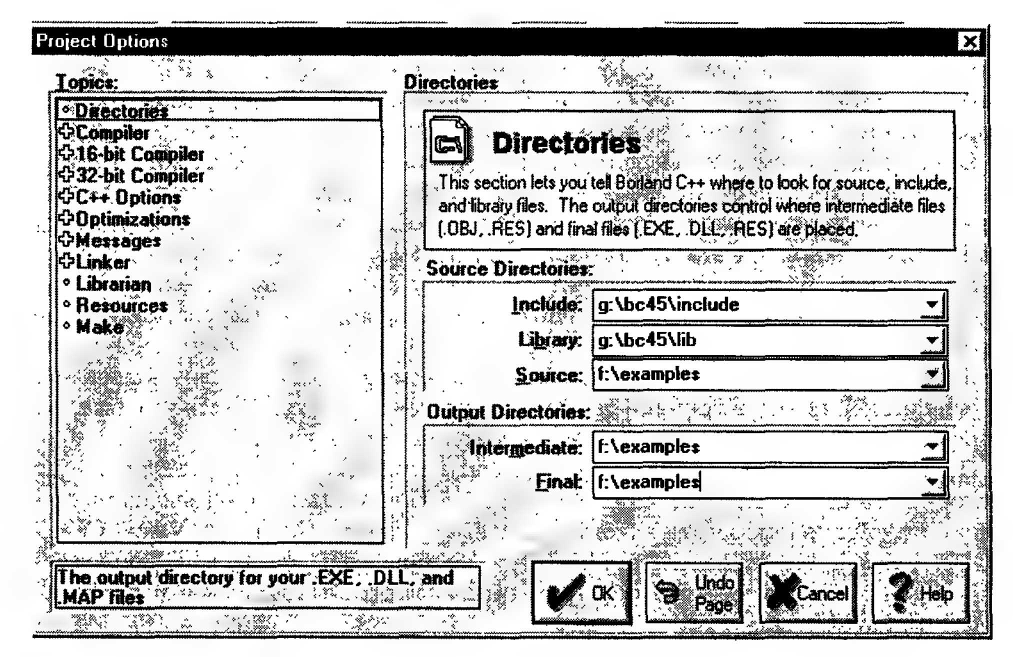


Рис. 76.2. Настройка каталогов пользователя

Далее полезно выбрать пункт Options➢Environment➢Editor➢Display и удостовериться, что установлен шрифт с русскими буквами (например, Courier New Cyr), и заодно задать удобный для пользователя размер шрифта. Если после выбора пункта Options➢Environment в перечнс пунктов следующего уровня пункт Editor окажется свсрнутым (о чем будет свидетельствовать знак + перед ним), его следуст развернуть, щелкнув по нему дважды левой клавишей мыши или щелкнув один раз по знаку +. Строка +Editor развернется в список

–Editor
- Options
- File
- Display

из которого уже можно выбрать пункт Display для настройки шрифта.

Для подготовки исходного текста программы выбсрите команду File➢New. Поскольку IDE даст новому файлу не очень благозвучное имя NONAME00.CPP и к тому же пытается расположить его в каталогс \BC45\BIN, лучшс всего сразу жс сохранить пустой пока файл под удобным для пользоватсля имснем в его рабочсм каталогс, что выполняется как обычно с помощью пункта мсню Filc➢Save As. Введитс исходный тскст приложсния Windows примера 76.1 и сохранитс сго на диске под имснсм, например, 76-01.CPP.

Как только на экранс IDE появляется окно для исходного текста программы, кадр IDE дополнястся цслым рядом новых управляющих кнопок, служащих для редактирования, запуска и отладки программы. Назначенис наиболсс важных кнопок указано на рис. 76.3.

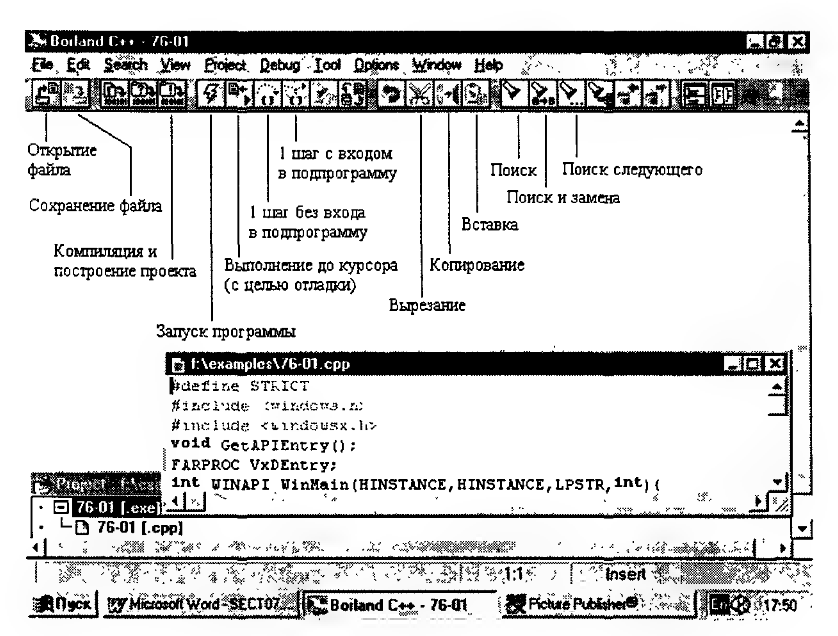


Рис. 76.3. Кадр IDE с текстом программы

Введя текст программы и сохранив его в файле с расширением .CPP, следует приступить к созданию проекта. Вообще проект служит для объединения в единый комплекс всех файлов, входящих в приложение. Обычно исходные тексты даже простого приложсния образуют не один, а по меньшей мере три файла: файл .CPP с текстом программы; файл ресурсов .RC, в котором, в частности, описывается конфигурация меню и диалоговых окон, и файл определения модуля .DEF с информацией о типе загрузочного модуля, атрибутах программных сегментов и некоторой другой информацией. Файл проекта (в системах Borland C++ 4.5 и 5.0 он имеет расширение .IDE) как раз и служит для объединения всех этих файлов в единый комплекс, относящийся к данному приложению.

Для нашей программы, в которой нет ни меню, ни диалогов, не нужен файл ресурсов; что же касается файла определений, то создавать его тоже нет необходимости, так как вполне можно воспользоваться стандартным файлом DEFAULT.DEF, имеющимся в системе. Однако создать проект все-таки необходимо, так как именно в проекте указываются некоторые важные характеристики приложения, без знания которых IDE не сможет создать загрузочный модуль приложения.

Выберите команду Project➢New Project и в верхней части открывшейся панели с именем New Target компонента IDE Target Expert в рамке Project Path and Name укажите имя файла проекта. Это имя может быть любым, но удобнее для всех составляющих проекта иметь одно имя (при разных расширениях). Поэтому укажите имя 76-01.IDE, при необходимости предварив его полным путем к вашему рабочему каталогу.

Удостоверьтесь, что в рамке Platform установлен режим Windows 3.x (16-битовое приложение Windows), а в рамке Target Model – Large (большая модель памяти). Выбор модели Large заставит компилятор рассматривать все адреса как дальние, что необходимо при программировании интерфейса с виртуальным драйвером.

Удостоверьтесь, что в рамке Target Type (тип мишени) установлен тип Application [exe], т. е. выполнимый файл приложения.

Нажмите кнопку Advanced. На открывшейся вкладке Advanced options снимите флажки (щелчком левой клавиши мыши) с кнопок выбора файлов .rc и .def, так как этих файлов у нас нет (рис. 76.4). Подтвердите настройки, выбрав OK на вкладке Advanced options и еще раз OK в панели Target Expert.

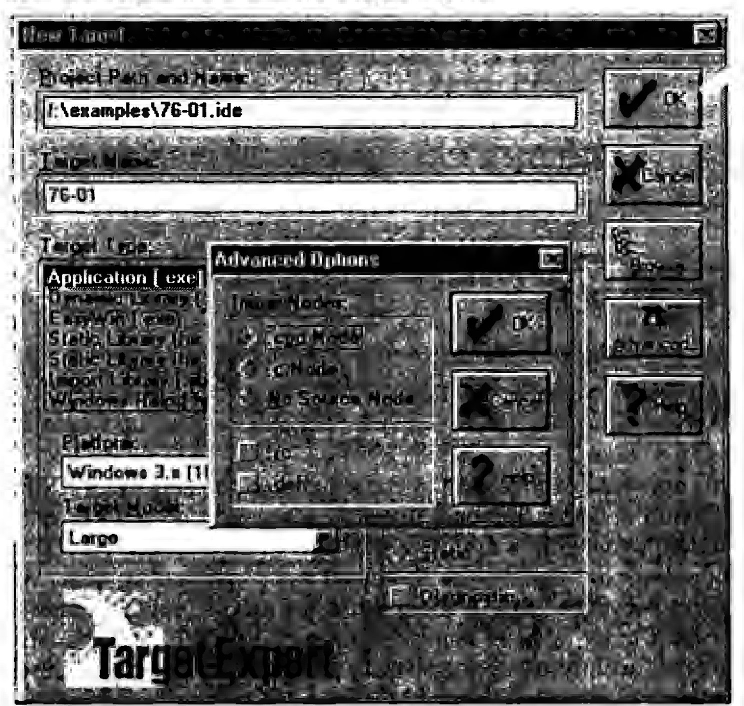


Рис. 76.4. Панель Target Expert с характеристиками приложения

В кадре IDE появится окно описания проекта Project (рис. 76.5). Видно, что наш проект будет состоять из двух файлов – 76-01.CPP и 76-01.EXE. В окне проекта можно при необходимости выполнять корректировку состава проекта (добавление новых файлов или удаление существующих), переименовывать отдельные составляющие проекта и изменять характеристики приложения. Для того чтобы в окне IDE появился исходный текст программы, щелкните дважды в окне проекта по строке 76-01 [.cpp]. Если же надо изменить какие-либо характеристики приложения (например, модель памяти), щелкните правой клавишей по строке 76-01 [.exe]. Откроется меню, из которого, в частности, можно выбрать пункт Target Expert и снова открыть окно, с которым мы уже имели дело.



Рис. 76.5. Открытое окно проекта

Для запуска процесса компиляции и компоновки щелкните по кнопке компиляции и построения проекта (см. рис. 76.3). Дождитесь окончания обработки программы и удостоверьтесь, что в окне построителя задачи сообщается об отсутствии ошибок (Errors: 0). Предупреждением, возникающим из-за того, что мы использовали файл DEFAULT.DEF определения по умолчанию, в данном случае можно пренебречь.

При повторной загрузке IDE Borland C++ с целью, например, модификации и исследования созданного приложения, необходимо прежде всего открыть файл проекта. Если просто открыть исходный файл программы и, не открывая файла проекта, попробовать выполнить компиляцию и запуск, вы получите сообщение об ошибках. Чтобы исправить дело, надо будет загрузить файл проекта и повторить компиляцию.

Перейдем, наконец, к содержательному рассмотрению текста приложения Windows (пример 76.1).

Пример 76.1. Взаимодействие драйвера и приложения. Исходный текст приложения Windows

```
#define STRICT             //(1)Более строгая проверка типов переменных
#include <windows.h>       //(2)Два файла с определениями констант, макросами
#include <windowsx.h>      //(3)и прототипами функций Windows
void GetAPIEntry();        //(4)Прототип функции
FARPROC VxDEntry;          //(5)Объявление переменной типа FARPROC
struct DESCR{              //(6)Описание структуры, повторяющей состав дескриптора
  WORD lim;                //(7)Граница (биты 0...15)
  WORD base_l;             //(8)База, биты 0...15
  BYTE base_m;             //(9)База, биты 16...23
  BYTE attr_1;             //(10)Байт атрибутов 1
  BYTE attr_2;             //(11)Граница (биты 16...19) и атрибуты 2
  BYTE base_h;             //(12)База, биты 24...31
  }dscr;                   //(13)Объявление переменной dscr типа структуры DESCR
//Главная функция WinMain
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){//(14)
  char txt[160];           //(15)Объявление текстовой строки
  GetAPIEntry();//(16)Вызов функции GetAPIEntry, получение адреса драйвера
  _DI=OFFSETOF(&dscr);     //(17)В драйвер будет передано смещение dscr
  VxDEntry();              //(18)Вызов API-процедуры драйвера
  wsprintf(txt,"Базовый линейный адрес = %X%X%Xh"  //(19)Подготовка строки
     "\nАтрибут 1 = %Xh   Атрибут 2 = %Xh",        //с текстом и данными
     "\nРазмер сегмента = %Xh=%d байт",            //для вывода на экран
     dscr.base_h,dscr.base_m,dscr.base_l,
     dscr.attr_1,dscr.attr_2,++dscr.lim,dscr.lim);
  MessageBox(NULL,txt,"Info",MB_ICONINFORMATION);//(20)Вывод сообщения
  return 0;                //(21)Завершение главной функции
  }                        //(22)Конец главной функции
//Функция GetAPIEntry() получения точки входа в драйвер
void GetAPIEntry(){        //(23)Описание функции GetAPIEntry
  asm{                     //(24)Вставка на языке ассемблера
      mov AX,0x1684        //(25)Номер функции
      mov BX,0x8000        //(26)Идентификатор драйвера
      int 0x2F             //(27)Мультиплексное прерывание
      mov word ptr VxDEntry,DI//(28)Смещение точки входа в драйвер
      mov word ptr VxDEntry+2,ES//(29)Селектор точки входа в драйвер
      }                    //(30)Конец ассемблерной вставки
  }                        //(31)Конец функции GetAPIEntry
```

При запуске приложения Windows управление всегда передается главной функции WinMain(), которая, таким образом, должна присутствовать в любой программе. Помимо главной функции, в программе может быть любое число вспомогательных функций-подпрограмм, вызываемых в тех или иных точках главной функции. В нашей про-

грамме имеется одна такая функция-подпрограмма GetAPIEntry(), служащая для определения адреса точки входа в драйвер. Для того чтобы по ходу изложения было понятно, о каком объекте идет речь – о функции или о переменной, мы будем обозначения функций сопровождать парой круглых скобок (), подчеркивая тем самым, что этому объекту в принципе присущи параметры.

Программа начинается с группы операторов (директив) препроцессора, обрабатываемых еще до компиляции текста программы. Предложение

```
#define STRICT
```

объявляет имя STRICT определенным (известным). В этом случае компилятор осуществляет более строгую проверку типов данных, используемых в программе, что дает возможность выявить и устранить многие ошибки еще на стадии компиляции. Для нашей программы, в которой используется минимальное число типов (BYTE, WORD, int и некоторые другие) эта проверка не имеет значения, однако в более сложных программах она весьма полезна.

В предложениях 2 и 3 к исходному тексту программы с помощью оператора препроцессора #include подсоединяются заголовочные файлы WINDOWS.H и WINDOWSX.H, в которых содержатся описания производных типов данных, констант, макросов и прототипов функций Windows. Например, использованные в программе типы данных BYTE и WORD являются типами не классического языка Си, а системы программирования для Windows. В языке Си им соответствуют типы unsigned char и unsigned short int. Определения специфических для системы Windows типов даны в файле WINDOWS.H. Там же можно найти определение использованного нами типа FARPROC как дальнего указателя на функцию, не требующую параметров и не возвращающую результата.

В языке Си действует правило: для всех функций, используемых в программе, должны быть приведены их прототипы. Прототип – это сокращенное определение функции, в которое входит имя функции, типы принимаемых ею параметров и тип возвращаемого результата. Например, функция Func(), выполняющая некоторую операцию над двумя целыми числами и возвращающая результат с плавающей запятой, будет иметь следующий прототип:

```
float Func(int,int);
```

Для функций, определенных в прикладной программе, прототип указывается в начале программы или, чаще, в "персональном" заголовочном файле, принадлежащем этой программе. Прототипы же стандартных функций Windows, в частности используемых в нашей программе функций WinMain() и MessageBox(), приведены в файле WINDOWS.H (а также и в других включаемых файлах с расширением .H). Таким образом, включение в программу файла WINDOWS.H обязательно.

Некоторые более специальные определения объектов Windows помещены в файл WINDOWSX.H. Строго говоря, в рассматриваемом примере этот файл не нужен, однако в более сложных программах он также необходим. Проще не задумываться над тем, нужен он или нет, а просто включать его во все приложения Windows.

В предложении 4 определяется прототип прикладной функции GetAPIEntry(), которая служит для получения адреса точки входа в драйвер. Эта функция не требует никаких параметров и не возвращает результата (полученный ею адрес драйвера она запишет непосредственно в поля данных, в переменную VxDEntry). Отсутствие параметров обозначается в прототипе пустой парой скобок; отсутствие возвращаемого ре-

зультата – словом void перед именем функции. Определение прототипа, как и любой оператор языка Си, заканчивается знаком точки с запятой (это правило не относится к операторам препроцессора).

В предложении 5 объявляется переменная VxDEntry типа FARPROC, т. е. типа дальнего указателя на функцию. Действительно, API-процедура драйвера выступает для нашей программы в качестве вызываемой функции, причем она безусловно не входит в состав программы и находится в другом сегменте, поэтому ее адрес должен быть дальним. Оба объекта – и прототип функции, и переменная VxDEntry – должны быть указаны перед началом главной функции WinMain().

Предложения 6...13 описывают структуру, соответствующую составу дескриптора памяти. С подобной структурой мы уже сталкивались в статье 65; она не является необходимой (вместо нее можно было просто ввести в программу массив из восьми 1-байтовых переменных), однако несколько повышает наглядность операций с отдельными полями дескриптора. Слово DESCR является придуманным нами типом этой структуры (тоже придуманной нами), имя же конкретной переменной, для которой отводится место в памяти, указывается после заключительной фигурной скобки определения структуры. В нашем случае эта переменная носит имя dscr. Стоит отметить, что в данном примере в одной конструкции языка объединяется и определение состава новой структуры DESCR, и объявление конкретной структурной переменной dscr. Часто эти действия разделяют, сначала определяя состав структуры и назначая имя этому новому типу данных и лишь далее в нужном месте программы объявляя одну или несколько структурных переменных этого типа.

Переменные, описанные перед именем главной функции программы, носят название глобальных. Это означает, что к ним можно обращаться из любых точек программы, в частности из ее внутренних функций. Переменные, объявленные внутри функции, являются локальными; они доступны только из этой функции и не видны ни во внешних функциях, ни даже во внутренних, вызываемых из данной функции. Полезно иметь в виду, что глобальные переменные размещаются в сегменте данных программы, адресуемом через сегментный регистр DS, а под локальные выделяется место на стеке и доступ к ним осуществляется через сегментный регистр SS. Правда, в большинстве случаев в 16-разрядных приложениях, написанных на языке Си, содержимое этих регистров совпадает; под все данные выделяется один сегмент памяти, в начале которого (в области меньших адресов) располагаются глобальные переменные, а в конце (в области больших адресов) – стек. Однако так бывает не всегда, и в нашем случае, где переменная dscr должна быть доступна драйверу, лучше определить ее как глобальную, чтобы компилятор разместил ее в сегменте, адресуемом через DS.

Описание любой функции языка Си начинается с ее заголовка, включающего имя функции, тип возвращаемого значения и (в скобках) перечень типов принимаемых функцией параметров с указанием их фактических обозначений внутри функции. Часто бывает так, что, хотя функция, в соответствии с объявленным ранее прототипом, требует при ее вызове передачи ей некоторых параметров, в конкретном контексте она эти параметры не использует. В этом случае в заголовке функции обозначения параметров можно опустить, хотя их типы должны быть перечислены обязательно. Именно так используется в настоящем примере главная функция WinMain(), которая при ее вызове системой Windows (в процессе запуска нашей программы) получает 4 параметра типов HINSTANCE, HINSTANCE, LPSTR и int. Программа в силу ее простоты эти

параметры не использует, и в заголовке функции указаны только их типы. В более сложной программе заголовок функции WinMain может выглядеть, например, так:

```
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE,LPSTR,int nCmdShow)
```

Здесь предполагается, что функция WinMain() будет использовать в процессе своего выполнения первый и последний из переданных ей системой параметров, а второй и третий использовать не будет.

Само описание функции, т. е. перечень выполняемых ею действий, заключается в фигурные скобки {}. Как уже упоминалось выше, каждый оператор, входящий в функцию, заканчивается знаком точки с запятой.

В предложении 15 объявляется символьная строка – массив, состоящий из 160 переменных типа char. В дальнейшем эта пока пустая строка будет заполнена упорядоченными данными из структуры dscr и использована для их отображения на экране.

С предложения 16 начинается собственно текст программы. Прежде всего вызовом

```
GetAPIEntry();
```

в переменную VxDEntry заносится полученный из Windows адрес точки входа в API-процедуру драйвера. Состав функции GetAPIEntry() и смысл возвращаемого ею значения мы рассмотрим чуть позже; на этом этапе удовольствуемся полученным результатом.

Получив адрес точки входа в драйвер, мы сможем вызвать его API-процедуру, которая должна вернуть 8-байтовое значение дескриптора памяти. Получить данные из виртуального драйвера (как и передать ему исходные данные) можно разными способами. В настоящей программе через регистры DS:DI драйверу передается полный адрес структурной переменной dscr, а драйвер, получив в процессе выполнения своей API-процедуры искомый дескриптор, записывает его значение непосредственно в переменную dscr приложения. Регистр DS уже содержит сегментный адрес сегмента данных программы; в предложении 17 в регистр DI с помощью макроса OFFSETOF заносится смещение структурной переменной dscr. В языке Си предусмотрена возможность непосредственного обращения ко всем регистрам общего назначения с помощью обозначений _AX, _AL, _DI и т. д. Следует обратить внимание на обозначение &ptr. Оператор &, встретившись перед именем переменной, обозначает ее адрес. Макрос OFFSETOF позволяет выделить младшую часть этого адреса, т. е. смещение.

Подготовив для драйвера исходные данные, вызовем его API-процедуру. Это делается в предложении 18, где адрес точки входа в драйвер VxDEntry рассматривается как имя дальней функции. После возврата из драйвера переменная dscr заполнена искомыми данными, и нам остается лишь вывести на экран ее содержимое в разумном виде. Это действие выполняется в два этапа. Сначала с помощью функции Windows wsprintf() значения полей переменной dscr преобразуются в символьную форму и помещаются в строку txt; затем эта строка выводится на экран в окно сообщения с помощью функции Windows MessageBox().

Если текст, выводимый в окно сообщения, известен заранее, его можно просто включить в вызов функции MessageBox() в качестве параметра:

```
MessageBox(NULL,'Драйвер отработал','Контроль', MB_ICONINFORMATION);
```

Если, однако, требуется вывести значения каких-либо числовых переменных, сначала эти значения следует преобразовать в символьную форму, для чего служит функ-

ция wsprintf(). Для этой функции первый параметр является адресом строки-приемника, второй – адресом строки-источника (или самой строкой, как в нашем примере), третий – преобразуемой переменной (или списком переменных, если их несколько). В строке-источнике, кроме текста, могут присутствовать спецификации формата, определяющие, по какому формату будут преобразовываться в символьные строки указанные в последнем параметре данные. Число указанных переменных (в нашем случае их 7: dscr.base_h, dscr.base_m, dscr.base_l, dscr.attr_1, dscr.attr_2 и два раза dscr.lim) должно совпадать с числом спецификаций формата. Все спецификации формата начинаются со знака процента (%); некоторые из них приведены в табл. 76.1.

Таблица 76.1. Допустимые спецификации формата для функции wsprintf() и их назначение

Спецификация	*Переменная*	*Представление при выводе*
d, i	Целое	Десятичное целое со знаком
U	Целое	Десятичное целое без знака
Lu	Длинное целое	Десятичное длинное целое без знака
x, X	Целое	16-ричное целое (строчные или прописные)
lx, lX	Длинное целое	16-ричное длинное целое
S	Строка	Строка
C	Байт	Символ

В нашем примере сначала из трех составляющих линейного адреса dscr.base_h, dscr.base_m и dscr.base_l формируется изображение одного длинного числа в 16-ричной форме, далее выводятся оба атрибута сегмента и, наконец, его длина сначала в 16-ричной форме (спецификация формата %Xh), а затем в десятичной (спецификация %d). Напомним, что в дескрипторе указывается граница сегмента, которая на 1 байт меньше его длины; для вывода на экран длины сегмента переменная dscr.lim включена в список переменных функции wsprintf() с префиксным оператором ++, который сначала увеличивает значение переменной на единицу и лишь затем передает ее по назначению. Второе включение той же переменной в список (уже без префиксного оператора) позволяет вывести в окно сообщения ее значение еще раз в другом (в данном случае десятичном) формате.

Функция Windows MessageBox(), выведя на экран окно сообщения, ожидает реакции пользователя; приложение продолжит свою работу после того, как пользователь щелкнет по клавише OK. В нашем примере вслед за вызовом функции MessageBox() стоит оператор return, который завершает работу приложения.

Рассмотрим теперь подпрограмму GetAPIEntry(), которая позволяет получить адрес точки входа в драйвер. Для взаимодействия с виртуальными драйверами в Windows предусмотрен ряд функций мультиплексного прерывания 2Fh. Например, с помощью функции 1681h можно сообщить драйверу о начале критической секции программы, а с помощью функции 1682h – о ее завершении. Для получения адреса API-процедуры используется функция 1684h, которой в регистре BX следует передать идентификационный номер нашего виртуального драйвера. Вызвать мультиплексное прерывание 2Fh (как и любое другое программное прерывание) можно с помощью функции Си int86x(); однако проще и нагляднее это сделать, использовав ассемблерную вставку. В функции GetAPIEntry после заполнения регистров AX и BX исходными значениями и вызова прерывания 2Fh результат выполнения функции 1684h, воз-

вращаемый в регистрах ES:DI, отправляется непосредственно в предусмотренную нами переменную VxDEntry.

Результат выполнения программы примера 76.1 (в предположении, что мы уже написали и установили соответствующий ему виртуальный драйвер) приведен на рис. 76.6.

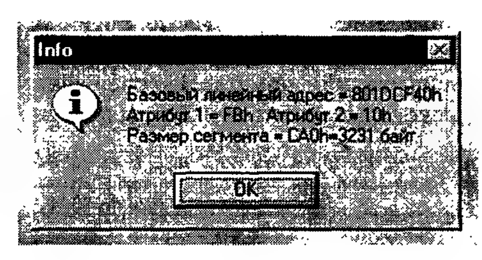


Рис. 76.6. Вывод программы

Перейдем теперь к тексту виртуального драйвера, работающего в паре с рассмотренной выше программой (пример 76.1, продолжение).

Пример 76.1 (продолжение). Взаимодействие драйвера и приложения. Исходный текст виртуального драйвера, пересылающего в приложение данные из таблицы локальных дескрипторов

```
.386p
.XLIST
include vmm.inc
.LIST
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h, \
    Undefined_Init_Order,,API_Handler
;Определим структуру descr, описывающую состав дескриптора
descr   struc
lim     dw 0
base_l  dw 0
base_m  db 0
attr_1  db 0
attr_2  db 0
base_h  db 0
descr   ends
;=============Сегмент инициализации реального режима==============
VxD_REAL_INIT_SEG
;Процедура инициализации реального режима
BeginProc VMyD_Real_Init
        mov    AH,09h
        mov    DX,offset msg
        int    21h
        mov    ax, Device_Load_Ok
        xor    bx,bx
        xor    si,si
        xor    edx,edx
        ret
msg db 'Виртуальный драйвер VmyD загружен'
EndProc VMyD_Real_Init
VxD_REAL_INIT_ENDS
;==============Сегмент данных=========================================
VxD_DATA_SEG    ,
pdescr  df 0             ;Поле для псевдодескриптора
d       descr <>         ;Поле для дескриптора
VxD_DATA_ENDS
;==============Сегмент защищенного режима===========================
VxD_CODE_SEG
;Управляющая процедура для обработки системных сообщений Windows
```

```
BeginProc VMyD_Control
        clc                   ;Мы не обрабатываем
        ret                   ;сообщения Windows
EndProc VMyD_Control
;Процедура, вызываемая из приложения защищенного режима (приложения Windows)
BeginProc API_Handler
;Получим из GDT дескриптор LDT
        sgdt   pdescr       ;(1)Линейный адрес и граница GDT
        mov    ESI,dword ptr pdescr+2;(2)Линейный адрес GDT
        sldt   DX           ;(3)EDX=Селектор LDT
        mov    EAX,[EDX][ESI];(4)Извлечем из GDT 1-ю половину дескриптора LDT
        mov    dword ptr d,EAX;(5)Отправим ее в d
        mov    EAX,[EDX+4][ESI];(6)Извлечем из GDT 2-ю половину дескриптора
        mov    dword ptr d+4,EAX;(7)Отправим ее в d
;Извлечем из дескриптора LDT линейный адрес LDT
        mov    EDX,dword ptr d.base_l;(8)3 байт базы + атрибут 1
        and    EDX,0FFFFFFh;(9)Уберем атрибут 1
        movzx  EAX,byte ptr d.base_h;(10)Старший байт базы в EAX
        shl    EAX,24       ;(11)Сдвинем его в старший байт EAX
        or     EDX,EAX      ;(12)EDX=линейный адрес LDT
;Получим из приложения селектор сегмента команд и найдем его дескриптор в LDT
        movzx  ECX,[EBP.Client_CS];(13)Селектор из CS приложения
        and    ECX,0FFFFFFF8h ;(14)Уберем 3 младших бита
        mov    EAX,[EDX][ECX] ;(15)Извлечем из GDT 1-ю половину дескриптора
                              ;приложения
        mov    dword ptr d,EAX;(16)Отправим ее в d
        mov    EAX,[EDX+4][ECX];(17)Извлечем из GDT 2-ю половину дескриптора
                              ;приложения
        mov    dword ptr d+4,EAX;(18)Отправим ее в d
;Перешлем полученный дескриптор из d в приложение
        Client_Ptr_Flat EDI,DS,DI;(19)EDI=линейный адрес переменной d приложения
        mov    EAX,dword ptr d;(20)1-я половина дескриптора
        mov    [EDI],EAX    ;(21)Отправим ее в приложение
        mov    EAX,dword ptr d+4;(22)2-я половина дескриптора
        mov    [EDI+4],EAX ;(23)Отправим ее в приложение
        ret                 ;(24)Возврат через Windows в приложение
EndProc API_Handler
VxD_CODE_ENDS
end VMyD_Real_Init
```

Общая структура драйвера почти не отличается от той, что была рассмотрена в примере 75.1. В блоке описания драйвера указана лишь одна API-процедура, именно процедура защищенного режима, так как мы не собираемся вызывать драйвер из программ DOS. Эта единственная процедура для краткости названа просто API_Handler. В сегменте данных появились поля для данных, используемых драйвером в процессе его работы. Поле pdescr размером 6 байт служит для размещения в нем содержимого регистра процессора GDTR, в котором, как известно, хранится линейный адрес и граница таблицы глобальных дескрипторов GDT. Второе поле d представляет собой структуру, полностью совпадающую со структурой dscr в приложении Windows и служащую для помещения в нее содержимого найденного дескриптора.

В одном из элементов глобальной таблицы дескрипторов GDT хранится системный дескриптор, описывающий локальную таблицу дескрипторов LDT данной задачи (рис. 76.7). Селектор этого дескриптора хранится в регистре LDTR и может быть прочитан оттуда командой sldt. Линейный адрес (вместе с границей) глобальной таблицы дескрипторов хранится в регистре GTDR, откуда его можно получить с помощью команды sgdt. Содержимое сегментного регистра CS нашего приложения представляет

собой селектор, указывающий на тот дескриптор в составе LDT, который описывает сегмент команд приложения. В этом дескрипторе и находятся интересующие нас данные, в частности линейный адрес сегмента команд и его предел. Приведенные на рис. 76.7 значения относятся к конкретному сеансу и носят случайный характер, хотя некоторые характеристики этих значений отражают закономерности системы Windows. Так, таблица глобальных дескрипторов вместе со многими другими системными компонентами располагается в 4-м гигабайте линейного адресного пространства, который начинается с адреса C0000000h, а 16-разрядные прикладные программы – в 3-м гигабайте с адреса 80000000h.

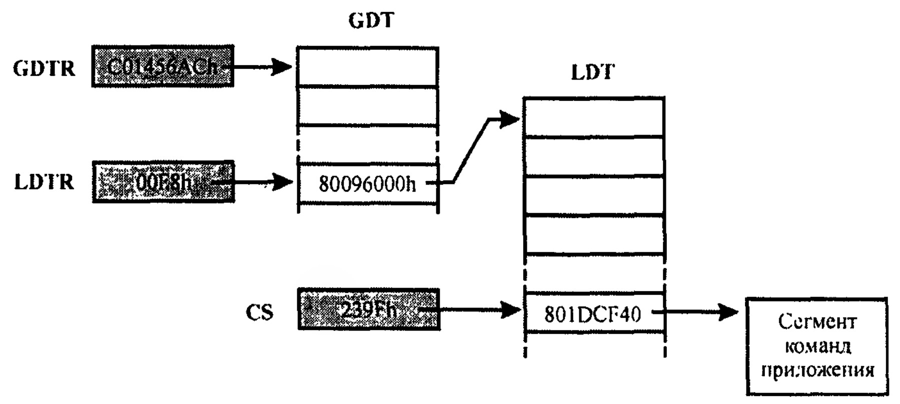


Рис. 76.7. Логическое взаимодействие системных регистров и таблиц, определяющих линейный адрес приложения

Вернемся к рассмотрению алгоритма API-процедуры виртуального драйвера (для удобства ссылок строки этой процедуры пронумерованы). В предложении 1 командой sgdt читается и запоминается в поле pdescr содержимое регистра GDTR. В байтах 0...1 этого регистра хранится значение границы GDT, в байтах 2...5 – базовый адрес. В предложении 2 базовый адрес извлекается из pdescr и запоминается в регистре ESI. Далее командой sldt в регистр DX заносится содержимое регистра LDTR, в котором хранится селектор LDT (в конкретном сеансе, проиллюстрированном на рис. 76.7, он оказался равен 00F8h). Селектор представляет собой смещение в байтах соответствующего дескриптора от начала таблицы; поскольку каждый дескриптор занимает 8 байт, дескриптор, описываемый селектором 00F8h, занимает F8h/8=1Fh=31-е место в таблице GDT (если начинать отсчет элементов GDT, как это и полагается, с нуля).

Линейный адрес интересующего нас дескриптора равен сумме базового адреса GDT и относительного адреса дескриптора в этой таблице. В нашей программе эти составляющие находятся в регистрах ESI и EDX (при заполнении регистра DX командой sldt автоматически очищается старшая половина EDX). В предложениях 4...7 две половины дескриптора переносятся в поле данных d.

Дескриптор, в котором хранятся характеристики локальной таблицы дескрипторов, относится к числу системных. Обобщенный формат системного дескриптора был приведен на рис. 70.1; ниже (рис. 76.8) он повторен с указанием конкретных значений для рассматриваемого прогона программы.

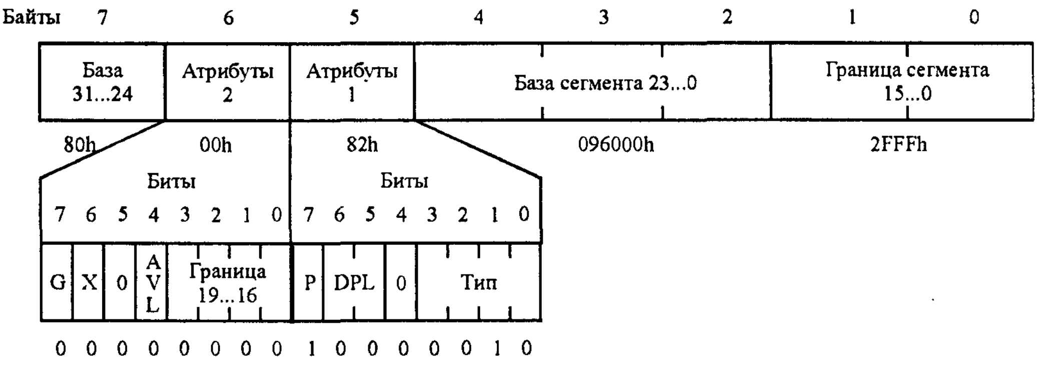


Рис. 76.8. Формат дескриптора LDT с конкретными значениями

Любопытно отметить, что таблица локальных дескрипторов располагается, как и 16-разрядное приложение Windows, в 3-м гигабайте линейных адресов и имеет довольно большой размер – 3000h байт = 12 Кбайт.

Нас в дескрипторе LDT интересует только линейный базовый адрес, который придется складывать из его составляющих в байтах 7, 4, 3 и 2 дескриптора. В предложении 8 байты 5...2 дескриптора загружаются в регистр EDX, а в предложении 9 командой and в нем очищается старший байт (в который попал байт атрибутов 1). Далее старший байт базы загружается в регистр EAX, сдвигается влево до конца регистра и командой or дописывается в EDX, где уже находятся 3 младших байта базы.

Смещения в LDT к дескрипторам сегментов памяти приложения (сегментов команд, данных и стека) определяются значениями селекторов этих сегментов, хранящихся в сегментных регистрах приложения. Однако, как уже отмечалось в статье 74, при переходе в виртуальный драйвер регистры процессора теряют те значения, которые они имели в приложении. Для получения "прикладных" значений регистров необходимо воспользоваться структурой клиента. В нашем примере структура клиента используется для получения драйвером селектора сегмента команд из регистра CS; передача результирующих данных в приложение осуществляется другим способом.

В предложении 13 API-процедуры селектор сегмента команд загружается в регистр ECX (с обнулением старшей половины регистра). Однако значение селектора не равно смещению в таблице дескрипторов, так как в его трех младших битах содержится дополнительная информация: индикатор таблицы дескрипторов TI и запрошенный уровень привилегий RPL (см. рис. 65.2). В нашем случае селектор сегмента команд имеет значение 239Fh; установленный бит 2 говорит о принадлежности этого селектора таблице локальных (а не глобальных) дескрипторов, а установленные биты 0 и 1 определяют уровень привилегий приложения, который, разумеется, равен трем. Для получения смещения следует обнулить 3 младших бита селектора, что и выполняется в предложении 14. К этому моменту в регистре EDX находится линейный адрес LDT, а в регистре ECX -- смещение к интересующему нас дескриптору. В предложениях 15...18 искомый дескриптор сегмента команд приложения переносится в то же поле данных d, которое мы ранее использовали для хранения дескриптора LDT.

Нам осталось переслать 8-байтовое поле d в приложение. Для этого можно воспользоваться структурой клиента, занеся в 4 элемента этой структуры, например Client_AX, Client_BX, Client_CX и Client_DX, 4 четверти дескриптора, а после возвра-

та в приложение извлечь дескриптор по частям из регистров AX, BX, CX и DX. Мы поступим иначе, передав весь дескриптор непосредственно в поле dscr сегмента данных приложения. Это можно сделать, так как виртуальный драйвер работает в плоской модели памяти и ему доступны все 4 Гбайт линейных адресов, в том числе и виртуальные адреса, назначенные системой приложению. Для перевода виртуального адреса в линейный в файле VMM.INC предусмотрен макрос Client_Ptr_Flat, в качестве параметров которого указывается 32-разрядный регистр-приемник, а также сегментный регистр и какой-либо регистр общего назначения, содержащие пару *сегмент:смещение* (для приложения реального режима) или *селектор:смещение* (для приложения защищенного режима).

Имея в виду это средство, мы перед вызовом драйвера занесли в регистр DI смещение переменной dscr; сегментный адрес сегмента данных приложения находится, естественно, в регистре DS. В предложении 19 API-процедуры драйвера с помощью макроса Client_Ptr_Flat в регистр EDI помещается линейный адрес переменной d, а в четырех последующих предложениях две половины дескриптора переносятся в поле dscr с использованием в качестве перевалочного пункта регистра EAX.

Завершающая API-процедуру команда ret передает управление через Windows в ту точку приложения, откуда был вызван драйвер.

Вернемся еще раз к макросу Client_Ptr_Flat. Каким образом он преобразует виртуальный адрес в линейный? Ведь макрос представляет собой просто некоторую последовательность директив или команд, имеющую имя и допускающую настройку под конкретные значения параметров. Однако никакими арифметическими или логическими операциями невозможно из величин *селектор:смещение* получить линейный адрес. Для определения линейного адреса по значению селектора необходимо обратиться к таблице локальных дескрипторов, найти в ней элемент, соответствующий данному селектору, извлечь из этого элемента составляющие базового линейного адреса сегмента и прибавить к нему смещение. Между прочим, именно эту операцию мы и выполняли в нашем примере виртуального драйвера. Таким образом, макрос Client_Ptr_Flat должен для образования линейного адреса обращаться к каким-то специальным системным средствам. Действительно, в тексте макроса имеется обращение к системной функции Map_Flat, которая извлекает из таблиц дескрипторов линейные адреса, выполняя, в сущности, те же операции, что и наша API-процедура. Для получения интересующего нас линейного адреса сегмента команд приложения вполне можно было воспользоваться этой функцией, существенно сократив размер API-процедуры. Правда, в этом случае мы не получили бы полной информации о составе дескриптора, а именно значений границы и атрибутов.

Статья 77. **Системный отладчик SoftICE**

Для исследований системы Windows, и в частности для отладки виртуальных драйверов, требуется специальный отладчик, работающий на нулевом уровне привилегий в плоском адресном пространстве. Одним из таких отладчиков является инструментальный пакет NuMega SoftICE от Compuware Corporation. Отладчик SoftICE позволяет изучать работу программ, работающих на любом уровне привилегий, выводить на экран поля данных и коды системных компонентов Windows, находить соответствие между виртуальными, линейными и физическими адресами, устанавли-

вать в отлаживаемой программе (в том числе в виртуальном драйвере) точки останова на определенных адресах, а также по аппаратным и программным прерываниям, выводить на печать фрагменты кодов прикладных и системных программ.

Отладчик SoftICE поступает к потребителю в виде дистрибутива и требует установки в системе, причем для систем Windows 95/98 или Windows NT и установка и использование отладчика осуществляются с некоторыми отличиями. В развернутом виде пакет SoftICE занимает около 16 Мбайт дискового пространства и содержит, кроме основных и вспомогательных программ, весьма подробный интерактивный справочник.

Пользователь взаимодействует в основном с двумя программами, входящими в состав SoftICE: WINICE.EXE и LOADER32.EXE. Первая программа устанавливается резидентной в системе в процессе загрузки Windows; вторая вызывается пользователем по мере необходимости. Для загрузки резидентной части SoftICE в файл AUTOEXEC.BAT следует включить строку

```
G:\SOFTICE\WINICE.EXE
```

в предположении, что пакет SoftICE находится в каталоге SOFTICE диска G:.

Запустить отладчик можно двумя способами. Первый заключается в активизации отладчика вне связи с какой-либо программой; он осуществляется нажатием комбинации клавиш Ctrl+D. В таком режиме можно, например, просматривать содержимое таблиц дескрипторов и таблиц трансляции, изучать характеристики действующих виртуальных машин, исследовать содержимое физической памяти. Для выхода из отладчика следует ввести команду отладчика X или еще раз Ctrl+D.

Второй способ заключается в запуске отлаживаемой программы под управлением отладчика. Для этого следует запустить программу LOAD32.EXE. На экране появится начальный кадр SoftICE (рис. 77.1), в котором надо сначала открыть файл с отлаживаемой программой с помощью самой левой кнопки, а затем загрузить его нажатием второй слева кнопки. Другие кнопки, как и меню, служат для настройки режимов работы SoftICE. Все эти операции подробно описаны в интерактивном справочнике, входящем в состав пакета; здесь мы отметим только одну полезную возможность.

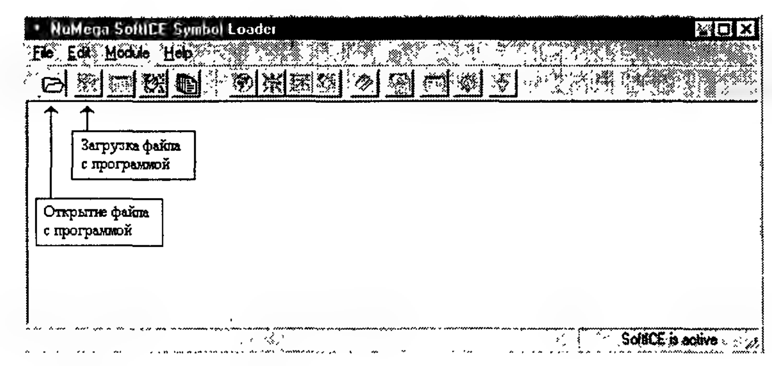


Рис. 77.1. Начальный кадр отладчика SoftICE

С помощью пункта меню Edit>SoftICE Initialization Settings можно изменить некоторые настройки отладчика, и в частности строку инициализации, определяющую исходный режим работы отладчика. По умолчанию строка инициализации состоит из одной команды X, оканчивающейся точкой с запятой. Эта команда автоматически завершает работу отладчика при его первой инициализации в процессе загрузки Windows. Если удалить команду X

из строки инициализации, то при перезагрузке Windows еще до загрузки виртуальных драйверов управление будет передано отладчику.

По умолчанию рабочий кадр с программой содержит всего 25 строк текста, при этом на экран не выводятся коды команд. Поскольку при отладке программы желательно иметь на экране максимум информации, строку инициализации полезно сформировать следующим образом:

```
LINES=60;CODE ON;X;
```

В этом случае на экран будут выводиться 60 строк текста, а команды отлаживаемой программы (так же как и команды драйвера или программных составляющих Windows) будут сопровождаться их кодами.

Использование программы SoftICE для отладки драйверов Windows осложняется тем, что отлаживаемой программой является, в сущности, приложение Windows, составленное на языке Си. Однако отладчик SoftICE, деассемблируя образ программы в памяти, выводит на экран ее текст в виде машинных кодов и соответствующих им команд языка ассемблера. Программист должен уметь находить соответствие между предложениями языка Си и соответствующими им машинными кодами (результатом трансляции). Для решения этой задачи проще всего воспользоваться отладчиком приложений Windows TDW.EXE, входящим в пакет Borland C++. Запустить этот отладчик можно из среды разработки IDE, открыв проект отлаживаемой программы и выбрав пункт меню Tool➢Turbo Debugger. На экране появится кадр отладчика с исходным текстом программы, практически совпадающий с тем, что был показан на рис. 4.1 (хотя текст программы будет состоять из предложений языка Си).

Будем считать, что мы отлаживаем программу примера 76.1, загрузочный модуль которой имеет имя 76-01.EXE. Загрузив отладчик TDW, поместим курсор на предложение вызова драйвера

```
VxDEntry();
```

и выполним программу до этой точки, нажав клавишу F4 (Here, сюда).

Для получения текста программы в машинных кодах перейдем в окно процессора, выбрав пункт меню View➢CPU. На экранс появится кадр, схожий с показанным на рис. 4.6 и содержащий внутренние окна для текста программы, содержимого регистров процессора, дампа памяти и стека. При этом в основное внутреннее окно выводятся не только машинные коды команд и результат их преобразования в команды языка ассемблера, но и исходные предложения на языке Си (рис. 77.2).

```
#76-01#18:  VxDEntry();
  cs:083F►FF1E8402       call   far [_VxDEntry]
#76-01#19:  wsprintf(txt,"Базовый линейный адрес
  cs:08C3 FF368802       push   word ptr [_dscr]
  cs:08C7 FF068802       inc    word ptr [_dscr]
  cs:08CB A18802         mov    ax,[_dscr]
  cs:08CE 50             push   ax
  cs:08CF A08E02         mov    al,[028E]
  cs:08D2 B400           mov    ah,00
  cs:08D4 50             push   ax
```

Рис. 77.2. Окно отладчика TDW с текстом программы 76-01.EXE после останова на вызове VxDEntry()

Мы видим, что вызов функции VxDEntry() прсобразован транслятором с языка Си в дальний косвснный вызов чсрез ячейку с адресом 0284h, причем эта команда имеет

смещение в сегменте команд, равное 0BF8h. Между прочим, при повторных запусках программы 76-01.EXE содержимое CS будет каждый раз назначаться системой Windows заново, однако состав программы и смещения всех ее элементов (команд или данных) будут оставаться неизменными.

Запомнив адрес интересующей нас точки программы, вызовем отладчик SoftICE. После загрузки отлаживаемой программы на экран выводится рабочий кадр отладчика с текстом отлаживаемой программы. Зная смещение требуемой точки останова, можно выполнить программу сразу до этого места, а затем продолжить выполнение в пошаговом режиме, что позволит изучить процесс перехода из прикладной программы в программу драйвера. На рис. 77.3 показан кадр отладчика SoftICE после загрузки программы 76-01.EXE и останова ее на команде дальнего вызова виртуального драйвера (в данном случае 2A07:08BF).

```
EAX=00001684 EBX=18078000 ECX=00000015 EDX=00010284 ESI=00000278
EDI=00000288 EBP=801F1706 ESP=00001662 EIP=000008BF o d I s z a P c
CS=2A07 DS=23CF SS=23CF ES=003B FS=0000 GS=0000 DS:00000284=003B03E4
---------------------------------------------------------------PROT16-
2A07:08BC  MOV   DI,0288
2A07:08BF  CALL  FAR[0284]
2A07:08C3  PUSH  WORD PTR[0288]
2A07:08C7  INC   WORD PTR[0288]
2A07:08CB  MOV   AX,[0288]
2A07:08CE  PUSH  AX
2A07:08CF  MOV   AL,[028E]
2A07:08D2  MOV   AH,00
2A07:08D4  PUSH  AX
2A07:08D5  MOV   AL,[028D]
2A07:08D8  MOV   AH,00
2A07:08DA  PUSH  AX
2A07:08DB  PUSH  WORD PTR[028A]
2A07:08DF  MOV   AL,[028C]
-------------------------------75-01(01)------------------------------
Break due to G (ET=15.65 milliseconds)
:LDT 2A07
Sel.  Type     Base      Limit     DPL  Attributes
2A07  Code16   8152DB60  00000C9F  3      RE
:
          Enter a command (H for help)
----------------------------------------------------------------------
```

Рис. 77.3. Рабочий кадр отладчика SoftICE

В верхней части кадра располагается окно регистров; средняя часть отведена под программные коды; нижняя часть представляет собой окно команд. В рассматриваемом примере после загрузки программы была введена команда G 08BF выполнения программы до указанного адреса, а затем – команда LDT 2A07 вывода элемента локальной таблицы дескрипторов, соответствующего селектору сегмента команд (который в данном сеансе оказался равен 2A07h).

Дальнейшее управление отладчиком осуществляется с помощью команд, подаваемых с клавиатуры. Полный список команд с пояснениями можно найти в интерактивном справочнике; в табл. 77.1 приведен перечень наиболее важных команд отладчика, иллюстрирующий его возможности.

Таблица 77.1. Наиболее употребительные команды отладчика SoftICE

Команда	*Назначение*
BPX *addr*	Установка точки останова по адресу *addr*
BPINT *intr*	Установка асинхронной точки останова по прерыванию с номером *intr*. Прерывание может быть как программным, так и аппаратным
BL	Вывод информации о тóчках останова с их номерами
BC *n*	Удаление точки останова с номером *n*
BC *	Удаление всех точек останова
D*size addr*	Вывод в нижнюю часть кадра содержимого ячеек виртуальной памяти начиная с адреса *addr*
PEEK*size addr*	Вывод в нижнюю часть кадра содержимого ячейки физической памяти по адресу *addr*
G *addr*	Выполнение программы до точки с адресом *addr*
F7	Выполнение программы до положения курсора
F8	Выполнение одной команды программы. Вход внутрь подпрограмм, выполнение циклов по шагам
F10	Выполнение одной команды программы. Подпрограммы и циклы выполняются как одна команда
X	Выполнение программы до конца с возвратом в начальный кадр SoftICE. Нажатием кнопки загрузки файла можно запустить текущую программу повторно с начала
HBOOT	Перезагрузка компьютера. Используется в случае "зависания" отладчика, что происходит (при наличии ошибок в отлаживаемой программе) довольно часто, например, при отказах страниц
PrintScreen	Вывод содержимого экрана на принтер (принтер должен иметь драйвер для работы в DOS)
LINES*n*	Установка размера экрана (в числе строк). Параметр *n* может принимать значения 25, 43, 50, 60
CODE ON/OFF	Включение/выключение вывода на экран машинных кодов команд программы
U *addr*	Деассемблирование, т. е. вывод на экран содержимого программных кодов начиная с адреса *addr*
GDT *sel*	Вывод содержимого GDT. При указании селектора *sel* вывод одного дескриптора, соответствующего этому селектору
LDT *sel*	Вывод содержимого LDT. При указании селектора *sel* вывод одного дескриптора, соответствующего этому селектору
IDT *intr*	Вывод содержимого IDT. При указании номера вектора *intr* вывод одного дескриптора, соответствующего этому вектору
TSS	Вывод содержимого сегмента состояния задачи
CPU	Вывод содержимого регистров процессора и информации о процессоре
VM	Вывод характерных данных о действующих виртуальных машинах
PHYS *paddr*	Вывод всех линейных адресов, отображаемых на указанный физический адрес *paddr*
I *port*	Ввод данного из порта *port*
O *port value*	Вывод данного *value* в порт *port*
R	Возврат в отладчик после невосстановимого сбоя (например, после отказа страницы)

Примечания:

1. Адрес *addr* может задаваться одним смещением, если имеется в виду текущий сегмент команд или данных, или в форме *seg:offs*, если речь идет о другом сегменте (например, в виртуальном драйвере или в системном компоненте Windows).

2. Спецификатор *size* указывает на формат вывода: по байтам (B), словам (W), двойным словам(D).

С помощью отладчика SoftICE нетрудно проследить механизм взаимодействия приложения с виртуальным драйвером. Рассмотрим некоторые поучительные детали этого механизма.

Как видно из рис. 77.3, команда перехода в виртуальный драйвер выглядит в машинных кодах следующим образом:

```
CS:08BF  call far[0284]
```

В двойное слово по адресу DS:0284h, которое соответствует переменной FARPROC VxDEntry нашего приложения Windows, по ходу выполнения программы было записано то, что мы называли адресом API-процедуры виртуального драйвера. В действительности это совсем не так. Если посмотреть (с помощью SoftICE или обычного отладчика из пакета Borland C++) содержимое ячейки 0284h, там окажется адрес вроде 003Bh:03ECh. Заглянув в сегмент с селектором 3Bh, мы увидим, что в нем нет ничего, кроме длинной последовательности команд int 30h, предоставляющих стандартный для Windows механизм перехода из 16-разрядного приложения защищенного режима в 32-разрядную программу уровня 0. Этот переход осуществляется через дескриптор 30h таблицы дескрипторов прерываний IDT, описывающий некоторую процедуру со смещением, например C0001B04h в том самом сегменте с селектором 28h, в котором работает VMM. Переход с помощью команды int 30h из приложения уровня 3 в исполняемый сегмент уровня 0 возможен потому, что, хотя при загрузке в CS селектора 0028h текущий уровень привилегий CPL оказывается равен нулю, что и обусловливает работу программ VMM на самом внутреннем кольце защиты, шлюз IDT содержит значение DPL=3, отчего этот шлюз оказывается доступен прикладным программам кольца 3.

Программный фрагмент VMM, на который происходит переход через вектор 30h, выполняет ряд системных проверок и настроек. В частности, выполняется процедура VMM под названием Simulate_Far_Ret, которая снимает со стека приложения два слова адреса возврата, помещенные туда командой call far, и копирует их в ячейки структуры клиента Client_CS и Client_IP. Это приведет при передаче управления из VMM в текущую виртуальную машину к "эмуляции дальнего возврата", т. е. к переходу по этому адресу, как если бы программа, вызванная приложением командой call far, выполнила команду ret far.

VMM, выполнив все эти операции, командой дальнего перехода jmp far передает управление на точку входа в API-процедуру нашего драйвера, который, таким образом, продолжает работу на нулевом уровне защиты при значениях сегментных регистров CS=28h и DS=SS=ES=FS=GS=30h.

Виртуальный драйвер, выполнив свои функции, командой ret возвращает управление VMM. Тот выполняет некоторые завершающие действия и активизирует текущую виртуальную машину, в которой выполняется наше приложение Windows. В этот момент и происходит эмуляция дальнего возврата с передачей управления назад в прикладную программу. Все эти действия VMM по обеспечению перехода в драйвер и

возврата в программу, т. е. фактически системные издержки, составляют около 100 команд. Такова цена использования в прикладной программе стандартного системного средства – виртуального драйвера.

Статья 78. **Драйвер для работы с физической памятью**

Как известно, подключение к компьютеру измерительной или управляющей аппаратуры выполняется одним из двух способов: с отображением на пространство портов (пространство ввода-вывода) и с отображением на общее с памятью адресное пространство. В первом случае за регистрами подключаемой аппаратуры закрепляется требуемое количество свободных адресов из пространства портов (например, 300h...308h), а программирование аппаратуры осуществляется исключительно командами in или out. Во втором случае регистрам аппаратуры выделяется свободный участок адресного пространства 1-го мегабайта (за пределами первых 640 Кбайт, занятых оперативной памятью), например область D0000h...D0320h; при программировании аппаратуры можно использовать любые команды процессора. Первый способ применяется для подключения относительно простой аппаратуры, для управления которой достаточно лишь небольшого количества регистров. Второй способ удобен в тех случаях, когда аппаратура содержит большое количество регистров или внутреннюю память, которую желательно сделать непосредственно адресуемой со стороны компьютера. Примером устройства такого рода является система КАМАК, широко используемая для автоматизации экспериментальных физических установок.

Программирование аппаратуры КАМАК или ей подобной (в плане подключения к компьютеру) в системе DOS осуществляется довольно просто. В один из сегментных регистров данных, обычно регистр ES, засылается сегментный адрес участка адресного пространства, на который настроен дешифратор адресов, включаемый в состав интерфейсной части аппаратуры:

```
mov   AX,0D000h
mov   ES,AX
```

После этого обращение к требуемому регистру осуществляется любой подходящей командой процессора (mov, test, or, and и т. д.) с указанием смещения этого регистра относительно базового адреса выделенного участка памяти:

```
mov   BX,32h       ;Задание смещения к конкретному регистру
mov   AX,ES:[BX]   ;Чтение из регистра 32h аппаратуры
mov   ES:[BX+2],CX;Запись в регистр 34h аппаратуры
```

Если говорить конкретно об аппаратуре КАМАК, то ее программирование требует выполнения некоторых дополнительных операций, в частности предварительного занесения в регистр команд и состояний КАМАК номера требуемой функции КАМАК (чтение или запись, чтение со сбросом, просто сброс, блокировка или разблокировка и т. д.), однако суть дела от этого не меняется – обращение к регистрам КАМАК осуществляется путем пересылки данных с указанием сегментного адреса выделенного участка физических адресов и смещения в пределах этого участка.

Приведенные выше операции нельзя выполнить в обычном 16-разрядном приложении Windows, которое работает в виртуальном адресном пространстве и не имеет доступа к физической памяти. Однако обращение к любому участку физической памя-

ти легко осуществить с помощью виртуального драйвера, который может вызовом соответствующих функций VMM отобразить заданный диапазон физических адресов на линейное адресное пространство и, более того, предоставить приложению виртуальный адрес в формате *селектор:смещение* для непосредственной работы с физической памятью.

Для отладки и исследования такого рода драйвера удобно воспользоваться ПЗУ BIOS, которое, как известно, расположено по фиксированным физическим адресам F000h:0000h...FFFFh. Содержимое любого байта ПЗУ BIOS легко прочитать с помощью какого-либо отладчика, например программы DEBUG, входящей в состав DOS, и проконтролировать таким образом результат применения виртуального драйвера. Пример 78.1 представляет собой программный комплекс (виртуальный драйвер и вызывающее его приложение Windows), в котором выполняется чтение нескольких байтов ПЗУ BIOS. Рассмотрим сначала текст приложения Windows.

Пример 78.1. Чтение из ПЗУ BIOS даты его выпуска. Исходный текст приложения Windows

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
void GetAPIEntry();
FARPROC VxDEntry;
char* ptr;
//Главная функция WinMain
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  char txt[80];
  GetAPIEntry();
  _DI=OFFSETOF(&ptr);
  _SI=0xF000;
  VxDEntry();
  for(int i=0;i<=7;i++)
    txt[i]=ptr[i+0xfff5];
  txt[i]=0;
  MessageBox(NULL,txt,"Info",MB_ICONINFORMATION);
  return 0;
  }
//Функция GetAPIEntry() получения точки входа в драйвер
  void GetAPIEntry(){
  asm{
        mov   AX,0x1684
        mov   BX,0x8000
        int   0x2F
        mov   word ptr VxDEntry,DI
        mov   word ptr VxDEntry+2,ES
        }
  }
```

По своей структуре и функционированию эта программа не отличается от программы примера 76.1. Она состоит из главной функции WinMain() и функции GetAPIEntry() получения точки входа в драйвер. В области глобальных переменных объявлена дополнительная переменная ptr типа "указатель на символы". В эту двухсловную переменную драйвер вернет созданный им виртуальный адрес ПЗУ BIOS, после чего приложение Windows, задавая то или иное смещение, сможет обращаться к любым байтам всей 64-килобатовой области адресов BIOS.

Получив с помощью вызова GetAPIEntry() адрес точки входа в драйвер, следует подготовить параметры для передачи драйверу. В регистрах DS:DI будет передан ад-

рес переменной ptr; в регистре SI – сегментный адрес той области памяти, которую мы хотим отобразить на линейные адреса. После вызова VxDEntry() в переменной ptr оказывается виртуальный адрес ПЗУ BIOS. По правилам языка Си, в этом случае ptr[0] обозначает самый первый байт этой области, ptr[1] – следующий байт, а ptr[0xFFF5] – байт со смещением 0xFFF5 от начала ПЗУ. В цикле из восьми шагов 8 байт ПЗУ начиная с адреса 0xFFF5 заносятся в символьную строку txt, в байт txt[9] записывается 0 и функцией MessageBox() строка txt выводится в окно сообщения. Завершающий нуль нужен для того, чтобы функция MessageBox() знала, какую часть строки txt следует вывести на экран. В языке Си символьные строки всегда заканчиваются двоичным (не символьным) нулем, который для функций, обслуживающих строки, является признаком конца строки.

На рис. 78.1 приведен вывод рассмотренной программы – содержимое ячеек ПЗУ BIOS со смещениями FFF5h...FFFCh, где хранится дата выпуска данного варианта BIOS.

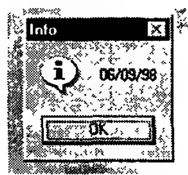


Рис. 78.1. Вывод программы

Перейдем теперь к рассмотрению виртуального драйвера, выполняющего отображение физических адресов на виртуальные. Хотя ниже приведен полный текст драйвера (пример 78.1, продолжение), фактически нам достаточно рассмотреть его API-процедуру.

Пример 78.1 (продолжение). Чтение из ПЗУ BIOS даты его выпуска.. Исходный текст виртуального драйвера

```
.386p
.XLIST
include vmm.inc
.LIST
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h, \
     Undefined_Init_Order,API_Handler,API_Handler
;=======================
VxD_REAL_INIT_SEG
BeginProc VMyD_Real_Init
        mov   AH,09h
        mov   DX,offset msg
        int   21h
        mov   ax, Device_Load_Ok
        xor   bx, bx
        xor   si, si
        xor   edx, edx
        ret
msg db 'Виртуальный драйвер VMyD загружен$'
EndProc   VMyD_Real_Init
VxD_REAL_INIT_ENDS
;======================
VxD_CODE_SEG
BeginProc VMyD_Control
        clc
        ret
EndProc VMyD_Control
BeginProc API_Handler
;Отобразим BIOS на линейные адреса
        movzx ESI,[EBP.Client_SI]
        shl   ESI,4
        push  0
        push  10000h
        push  ESI
```

```
        VMMCall _MapPhysToLinear
        add    ESP,3*4
;Получим селектор:смещение для начала BIOS
        mov    ECX,10000h
        VMMCall Map_Lin_To_VM_Addr;CX:EDX=sel:offs
        shl    ECX,16       ;Селектор в старшей части ECX
        or     EDX,ECX      ;EDX=sel:offs для обращения к BIOS
        Client_Ptr_Flat EDI,DS,DI;Получили адрес указателя из приложения
        mov    [EDI],EDX
        ret
EndProc API_Handler
VxD_CODE_ENDS
end VMyD_Real_Init
```

В этом драйвере не оказалось полей данных (которые присутствовали в примере 76.1), поэтому из него удален сегмент данных.

Для отображения физических адресов на линейные служит функция VMM _MapPhysToLinear. В справочнике DDK приведен следующий формат вызова этой функции:

```
        VMMCall _MapPhysToLinear, <PhysAddr,nBytes,flags>
```

где PhysAddr представляет собой базовый физический адрес отображаемого участка памяти, nBytes – размер отображаемого участка в байтах, а слово флагов flags должно иметь нулевое значение. Такая форма вызова функции VMM предполагает указание параметров в виде конкретных чисел. Однако для нас желательно иметь универсальный драйвер, которому можно было бы передавать базовый физический адрес в виде параметра. Дело в том, что аппаратура, адреса которой располагаются в пространстве памяти, всегда имеет средства настройки базового адреса, что позволяет устранять возможное наложение адресов, если к компьютеру подключено несколько таких установок. Поэтому драйвер, обслуживающий эту аппаратуру, также должен иметь средства настройки базового отображаемого адреса.

Обозначение VMMCall _MapPhysToLinear, несмотря на внешний вид напоминающий вызов функции VMM, представляет собой по существу макрос. Если в программе этот макрос используется с перечислением фактических параметров, ассемблер расширяет его приблизительно следующим образом:

```
        push   flags
        push   nBytes
        push   PhysAddr
        int    20h
dd      @@MapPhysToLinear ;0001006Ch
        add    ESP,12
```

Перечисленные в макровызове фактические параметры проталкиваются в стек, после чего вызывается прерывание 20h с указанием в следующем слове программы условного кода, соответствующего функции VMM MapPhysToLinear. Нетрудно сообразить, что системный обработчик прерывания 20h должен считать из программы код, следующий за командой прерывания, и передать управление на соответствующую процедуру VMM, которая в данном случае выделяет свободный диапазон линейных адресов и заполняет новые элементы таблицы трансляции, если указанные физические адреса еще не отображены на линейное адресное пространство. В процессе выполнения этой процедуры используются параметры, находящиеся в стеке, однако указатель стека не смещается. Восстановление стека (путем прибавления к ESP общей длины пе-

реданных параметров) происходит уже в драйвере после возврата из VMM последней командой макрорасширения.

Если макрос VMMCall _MapPhysToLinear вызывается без параметров, то в его макрорасширении отсутствуют как команды проталкивания в стек, так и команда восстановления стека. Поэтому обе эти операции следует включить в программу драйвера явным образом:

```
push   0            ;Имитация макроса с параметрами -
push   10000h       ;проталкивание параметров
push   ESI          ;в стек
VMMCall _MapPhysToLinear
add    ESP,3*4      ;Восстановление стека
```

Поскольку в число параметров функции MapPhysToLinear входит базовый физический адрес, а для нас естественнее передать в драйвер базовый сегментный адрес, в начале API-процедуры драйвера этот параметр переносится из структуры клиента в регистр ESI и сдвигается влево на 4 бита для образования физического адреса. Диапазон отображения (10000h = 64 Кбайт) задается в драйвере, хотя при необходимости его также можно было передать в качестве параметра через один из регистров.

Макрос VMMCall _MapPhysToLinear возвращает линейный адрес в регистре EAX (в случае ошибки возвращается код FFFFFFFFh). Второй этап преобразования – образование виртуального адреса, соответствующего полученному линейному, осуществляется с помощью вызова функции Map_Lin_To_VM_Addr (обратите внимание на отсутствие символа подчеркивания перед именем функции). Функция Map_Lin_To_VM _Addr требует в качестве параметров наличия в EAX преобразуемого линейного адреса, а в ECX – предела создаваемого сегмента. В нашем случае линейный адрес уже находится в EAX, остается только заслать границу сегмента – число FFFFh – в регистр ECX.

Функция Map_Lin_To_VM_Addr возвращает в регистре CX селектор образованного сегмента, а в регистре EDX – смещение к заданному линейному адресу, которое, впрочем, в приложениях защищенного режима всегда равно нулю. Полученный селектор смещается в старшую половину ECX и объединяется со смещением с образованием в регистре EDX обычного для 16-разрядного приложения формата дальнего адреса (селектор в старшем слове адреса, смещение в младшем). Следует заметить, что для приложения Windows эта операция является лишней, так как смещение всегда равно нулю; мы включили ее в драйвер для общности.

В последних предложениях API-процедуры драйвера с помощью макроса Client_Ptr_Flat формируется (в регистре EDI) линейный адрес поля данных ptr в приложении Windows, после чего полученный ранее виртуальный адрес физической памяти пересылается непосредственно в поле ptr. Приложение Windows, получив этот адрес, может обращаться ко всем отображенным физическим адресам, как это описывалось выше.

Статья 79. **Ввод-вывод через пространство портов**

В большинстве случаев подключение к компьютеру нестандартного внешнего устройства (например, автоматизируемой установки) осуществляется путем разработки для этого устройства интерфейсной платы, вставляемой в свободный разъем расширения ("слот"). Со стороны системной магистрали интерфейсная плата должна поддерживать стандартные протоколы обмена данными (операции ввода и вывода, а также,

возможно, протокол прерываний), а со стороны, обращенной к устройству, формировать необходимые для управления устройством сигналы и данные. Такая интерфейсная плата обычно имеет в своем составе ограниченный набор регистров управления и данных (чаще всего 2–3), которым с помощью дешифратора адреса, установленного на плате, назначаются определенные адреса из области ввода-вывода (0000h...FFFFh). Чаще всего эти адреса (порты) лежат в диапазоне 000h...3FFh. Разумеется, адреса, назначенные устройству, не должны совпадать с адресами штатных устройств компьютера. Программирование устройства выполняется в этом случае с помощью команд ввода-вывода in и out, а также их разновидностей.

Реакция процессора Intel на команды ввода-вывода обращения к тому или иному порту зависит, прежде всего, от соотношения уровня привилегий ввода-вывода и текущего уровня привилегий (CPL) выполняемого сегмента приложения, в котором встретилась команда ввода-вывода. Уровень привилегий ввода-вывода определяется значением поля IOPL в регистре флагов процессора и при выполнении Windows-приложений всегда равен нулю. Текущий уровень привилегий CPL задается значением двух младших битов в селекторе сегментов и для виртуальных драйверов (в частности, разработанных пользователем и включенных в систему) равен нулю, а для обычных Windows-приложений – трем.

Если CPL=IOPL (случай виртуального драйвера), команды ввода-вывода выполняются процессором обычным образом, путем непосредственного обращения к порту. Если же CPL>IOPL (случай обычного приложения), то процессор, встретив команду ввода-вывода, обращается к карте разрешения ввода-вывода в сегменте состояния задачи TSS. Эта карта содержит по 1 биту на каждый адрес из пространства ввода-вывода и занимает, таким образом, объем 64 Кбит = 8 Кбайт. Каждый бит карты может быть в отдельности сброшен (обращение к данному порту разрешено) или установлен (обращение к порту запрещено). В случае CPL>IOPL процессор, выполняя команду ввода-вывода для некоторого порта, анализирует состояние соответствующего ему бита в карте разрешения ввода-вывода и при нулевом значении этого бита выполняет непосредственное обращение к порту, а при единичном – формирует исключение общей защиты с номером 13 = 0Dh. Дальнейшая судьба затребованной операции ввода-вывода определяется алгоритмом работы обработчика исключения общей защиты, являющегося одним из элементов VMM, а также наличием или отсутствием виртуального драйвера для данного порта.

Карта разрешения ввода-вывода, формируемая Windows при загрузке системы, содержит единицы почти во всех битах. Открыты лишь порты 70h и 71h (КМОП-микросхема), 378h...37Fh (LPT1), 3F8h...3FEh (COM1) и некоторые другие. Таким образом, обращение практически к любому порту порождает исключение общей защиты, передающее управление системному обработчику этого исключения. Этот обработчик представляет собой 32-битовую программу уровня 0, расположенную в 4-м гигабайте линейного адресного пространства по адресу, например, 0028h:C00012B0h.

Дескриптор 0Dh таблицы дескрипторов прерываний IDT (как и многие другие дескрипторы этой таблицы) имеет атрибут 8Eh, что означает: присутствие, DPL=0, шлюз прерывания. Поскольку приложение выполняется на уровне 3, исключение предполагает переход на внутренний уровень 0, что усложняет процедуру прерывания. Процессор, реализуя эту процедуру, прежде всего переходит на стек уровня 0, кадр которого (значения SS:ESP) берется из TSS (поля со смещениями 4 и 8). Далее в стеке уровня 0

сохраняются значения SS:ESP приложения в точке прерывания, флаги процессора, значения CS:EIP (в данном случае – адрес еще не выполненной команды ввода-вывода) и, наконец, код ошибки. После этого в CS:EIP загружается адрес обработчика исключения 0Dh, находящийся в дескрипторе IDT, и начинает выполняться программа этого обработчика, которая прежде всего с помощью команды pushad формирует на стеке уровня 0 структуру регистров клиента (рис. 79.1), после чего переходит к анализу причины исключения.

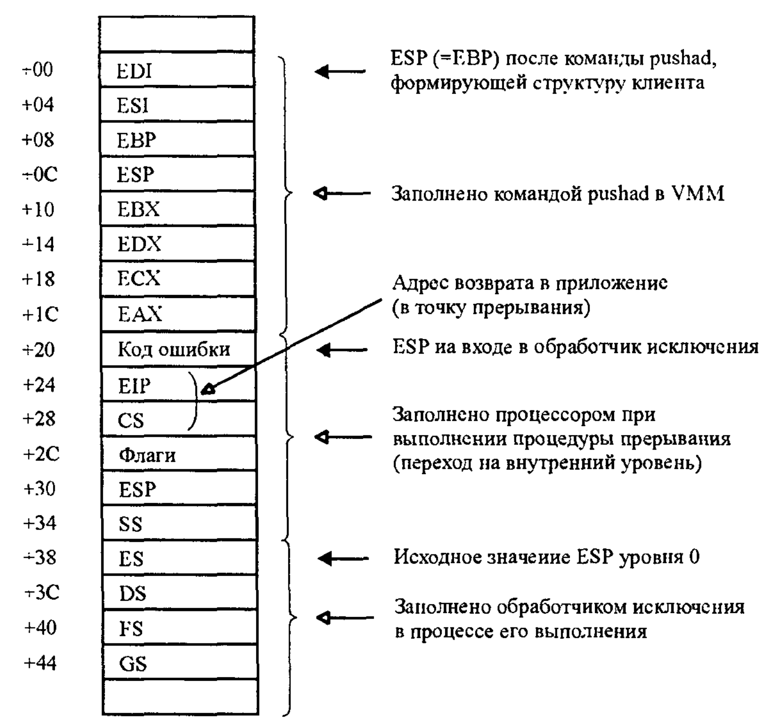


Рис. 79.1. Стек уровня 0 после перехода на уровень 0 в результате исключения

Исключение общей защиты может произойти в силу самых разнообразных причин: выхода за пределы сегмента, засылки в сегментный регистр несуществующего селектора, выполнения команды int с номером, отсутствующим в таблице дескрипторов прерываний и т. д. Обработчик исключения общей защиты определяет причину исключения, анализируя код команды, вызвавшей это исключение. Поскольку в стеке уровня 0, на котором работает обработчик, хранится адрес этой команды (значения CS:EIP), обработчик имеет возможность "залезть" по этому адресу в приложение, прочитать код команды и определить ход своих дальнейших действий.

Обработка исключения от команд in или out выполняется схожим образом, хотя и с некоторыми отличиями. Прежде всего над номером порта выполняется последовательность операций побитового сдвига и сложения по модулю 2, приводящая к образованию псевдослучайного числа, не превышающего 2FEh. Это число в дальнейшем используется в качестве индекса в таблицах, хранящих информацию о портах. Такой метод индексирования (называемый хешированием) позволяет существенно сократить

объем этих таблиц, но порождает проблему неоднозначности: каждому псевдослучайному числу может соответствовать несколько портов. Например, в число 2 преобразуются номера портов 80h, 100h, 281h и 301h, а в число C0h – номера портов 20h, 1A0h, 221h и 3A1h.

Используя полученное псевдослучайное число в качестве индекса, VMM обращается к таблице 2-байтовых величин, в которой хранится информация о наличии в системе виртуального драйвера для каждого порта. Если для данного порта виртуальный драйвер установлен, в соответствующей порту ячейке таблицы записан номер этого порта. Если драйвера нет, в ячейке записан 0. Заполнение таблицы номерами обслуживаемых Windows портов выполняется на этапе установки виртуальных драйверов в процессе загрузки системы.

В составе VMM имеется еще одна таблица (4-байтовых величин), адресация которой осуществляется с помощью тех же псевдослучайных чисел, умноженных предварительно на 2. В этой таблице хранятся адреса процедур виртуальных драйверов, предназначенных для обслуживания конкретных портов. Для свободных портов, не используемых Windows, в соответствующих ячейках этой таблицы хранится адрес процедуры обслуживания портов по умолчанию. Именно эта процедура будет вызвана при первом обращении к порту нестандартного внешнего устройства, если для него предварительно не написан виртуальный драйвер.

Обнаружив в ячейке первой таблицы номер искомого порта, VMM извлекает из второй таблицы адрес его драйвера и передает ему управление. Процедура виртуального драйвера, соответствующая данному порту, может, например, просто выполнить затребованную команду (in или out) и вернуть управление в приложение на команду, следующую за обработанной командой ввода-вывода. Поскольку VMM работает на нулевом уровне привилегий, он может обращаться ко всем портам, не вызывая нарушения общей защиты, даже если для данного порта установлен бит карты разрешения ввода-вывода в TSS.

Если в адресуемой ячейке первой таблицы не обнаружен номер искомого порта, это может означать, что либо для данного порта нет виртуального драйвера (но тогда в соответствующей ячейке второй таблицы должен храниться адрес процедуры обслуживания портов по умолчанию, которой и следует передать управление), либо в этой ячейке записана информация о каком-либо другом порте, номер которого преобразуется в то же псевдослучайное число. В этом случае выполняется анализ содержимого второй таблицы. Если там находится адрес процедуры по умолчанию (а это значит, что для данного порта виртуальный драйвер отсутствует), управление передается этой процедуре. Если же в ячейке второй таблицы иной адрес (т. е., видимо, адрес виртуального драйвера другого порта, отображаемого на ту же ячейку таблицы), выполняется декремент псевдослучайного числа и анализ содержимого предыдущих ячеек обеих таблиц. Эта процедура смещения по ячейкам таблиц влево повторяется до тех пор, пока в первой таблице не будет найден номер искомого порта или во второй таблице не будет обнаружен адрес процедуры по умолчанию.

В результате нарушение общей защиты, возбужденное командами in или out, приводит к передаче управления либо на виртуальный драйвер соответствующего порта, либо на процедуру обслуживания портов по умолчанию.

В процедуре по умолчанию выполняется целый ряд действий, из которых самым важным для нас является сброс бита карты разрешения ввода-вывода в TSS, соответствующего обслуживаемому порту. Эта операция выполняется командой

```
        btr    [tss+68h],EDX
```

перед выполнением которой в регистр EDX помещается номер адресуемого порта. Используемое в команде смещение представляет собой линейный адрес карты защиты ввода-вывода в TSS. Как известно, эта карта начинается в TSS со смещения 68h (см. рис. 70.3). Любопытно, что первым операндом этой команды в такой редакции является не регистр или ячейка памяти, а огромное поле памяти объемом 8 Кбайт, в котором с помощью второго операнда отыскивается и сбрасывается требуемый бит.

Сброшенное состояние бита сохраняется до перезагрузки Windows. Таким образом, в дальнейшем команды ввода-вывода с обращением к этому порту будут выполняться процессором непосредственно, без возбуждения нарушения общей защиты.

Выполнив коррекцию карты разрешения ввода-вывода в TSS, VMM устанавливает в структуре клиента значение EIP на адрес обрабатываемой команды ввода-вывода (в процессе обработки EIP был смещен на адрес следующей команды) и, восстановив из структуры клиента значения всех регистров процессора, с помощью команды дальнего возврата из прерывания iretd передает управление в приложение на ту же, еще не выполненную команду ввода-вывода. Поскольку данный порт теперь открыт, команда выполняется без помех.

Таким образом, программное обращение к портам, не используемым Windows, можно выполнять обычным образом, с помощью команд in и out, без каких-либо дополнительных программных средств. При этом, однако, следует иметь в виду, что первое обращение в любому порту будет проходить через обработчик нарушения общей защиты, что приведет к издержкам, составляющим несколько сотен команд. Последующие команды ввода-вывода будут выполняться непосредственно, без задержек. При необходимости можно включить в процедуру инициализации устройства фиктивные команды обращения ко всем его портам, чтобы открыть к ним доступ в карте TSS. Можно также написать и включить в систему виртуальный драйвер, который на этапе инициализации системы (или установки) снимет маску с требуемых портов в TSS. Наконец, можно включить в систему "полновесный" виртуальный драйвер, который возьмет на себя выполнение команд ввода-вывода для требуемых портов. В этом случае команды in и out будут по-прежнему вызывать нарушение общей защиты, однако оно будет обрабатываться не процедурой по умолчанию (снимающей маску в TSS), а прикладным виртуальным драйвером. Хотя такой метод наиболее соответствует идеологии Windows, однако для него характерны максимальные временные издержки и отсутствие видимых преимуществ по сравнению с прямым программированием через порты.

Обращение к порту в 16-разрядном приложении Windows можно выполнить с помощью функций Си inportb(), outportb(), прототипы которых описаны в заголовочном файле dos.h:

```
#include <dos.h>
outportb (0x300,0x71);
unsigned char data=inportb(0x301);
```

В этом фрагменте функцией Си outportb() в порт 300h выводится число 71h (предположительно команда управления устройством). В следующем предложении функци-

ей inportb() из порта 301h вводится данное. Принимающая переменная должна быть объявлена в этом случае как символьная без знака или типа BYTE. Предусмотрены и другие варианты функций ввода-вывода через порты, например функция inport(*port*), которая сразу вводит целое слово: младший байт из указанного порта *port* и старший байт из порта *port* + 1.

При программировании портов в 32-разрядных приложениях Windows возникает некоторая сложность из-за отсутствия функций вида inport(), outport(). Однако обращение к командам ввода-вывода нетрудно оформить в виде ассемблерных вставок. В примере 79.1 выполняется ожидание установки бита 7 в порту 30Ah, который сигнализирует о готовности данных в выходных портах установки 308h (младший байт данных) и 309h (старший байт). После получения сигнала готовности выполняется чтение данных из портов 308h и 309h и вывод этих данных в десятичной форме в окно сообщения.

Пример 79.1. Фрагмент 32-разрядного приложения Windows, выполняющего прямое обращение к портам

```
unsigned short int result; //Ячейка для результата
char txt[80];              //Символьная строка-буфер
...
waitx:                     //Метка
//Будем ожидать установки бита 7 в порту 30Ah
asm{                       //Начало ассемблерной вставки
  mov DX,0x30A             //Номер порта состояния
  in AL,DX                 //Ввод из порта
  test AL,0x80             //Анализ бита 7
  je waitx                 //Если 0, ждать дальше
  }                        //Конец ассемблерной вставки
//Прочитаем данные из установки
asm{                       //Начало новой вставки
  mov  DX,0x309            //Порт данных (старший байт)
  in   AL,DX               //Ввод старшего байта
  mov  AH,AL               //В AH его
  mov  DX,0x308            //Порт данных (младший байт)
  in   AL,DX               //Ввод младшего байта
  }                        //Конец вставки
result=_AX;                //Поместим результат в переменную
wsprintf(txt,"Накоплено %d событий",result);//Преобразуем данное в символы
MessageBox(NULL,txt,"Данные",MB_ICONSTOP);//Окно сообщения
}
```

Приведенный фрагмент взят из реальной программы управления измерительной установкой; вывод этой программы после завершения измерений показан на рис. 79.2.

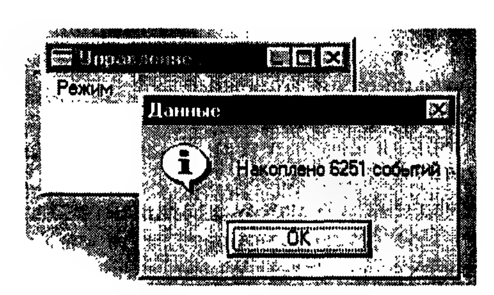


Рис. 79.2. Вывод 32-разрядной программы управления установкой

Статья 80. **Обработка аппаратных прерываний в системе Windows**

Автоматизированные установки, построенные на базе персонального компьютера, как правило, используют режим прерываний. Прерывания могут возбуждаться в установке по самым разным причинам: накопления заданного объема данных, выхода параметров установки за назначенные пределы, регистрации ожидаемого события и т. д. В любом случае приход прерывания должен активизировать в приложении определенную процедуру – обработчик прерываний, в которой выполняется то или иное обслуживание установки: прием накопленной информации, включение или выключение управляющих элементов, изменение режима работы установки и пр.

Выходы сигналов прерываний из установки подключаются к свободным уровням прерываний, а в программное обеспечение включаются обработчики прерываний соответствующих этим уровням векторов. Обычно используются уровни 3 или 5 ведущего контроллера (векторы 0Bh и 0Dh реального режима) или уровни 9...13 и 15 ведомого контроллера (векторы 71h...75h и 77h).

При работе в операционной системе MS-DOS контроллеры прерываний программируются так, что ведущему контроллеру назначается базовый вектор 8, а ведомому – 70h. Однако такая схема не может использоваться в защищенном режиме, так как первые 18 векторов заняты исключениями и не могут использоваться аппаратными прерываниями. Поэтому при загрузке Windows контроллеры прерываний программируются иначе: ведущему контроллеру назначается базовый вектор 50h, ведомому – 58h. Эти векторы отвечают последним 16 дескрипторам таблицы дескрипторов прерываний IDT, содержащей всего 60h дескрипторов.

В дескрипторах IDT, соответствующих аппаратным прерываниям, хранятся адреса системных обработчиков, принадлежащих виртуальному драйверу контроллера прерываний (виртуальному контроллеру) VPICD. Эти адреса назначаются системой, не могут быть изменены пользователем и вообще не видны из приложения. Если программист включил в 16-разрядное приложение Windows собственный обработчик какого-либо аппаратного прерывания, этот обработчик устанавливается с помощью функций Си getvect() и setvect(), которые преобразуются компилятором в вызовы функций 35h и 25h DOS. При вызове этих функций следует указывать номера векторов, которые использовались в DOS (8 для таймера, 70h для часов реального времени и т. д.). Вызовы этих функций в защищенном режиме перехватываются VPICD, который записывает адреса прикладных обработчиков не в IDT, а в отдельную таблицу векторов защищенного режима, которая создается в каждой виртуальной машине (если их несколько). В случае прихода аппаратного прерывания процессор вызывает его обработчик в соответствии с содержимым IDT. Этот обработчик (входящий в состав VPICD), обслуживая прерывание по правилам Windows, в конце концов передает управление по адресу, хранящемуся в таблице векторов защищенного режима.

Соответствие номеров векторов, номеров уровней прерываний и устройств, характерное для системы Windows, приведено в табл. 80.1.

Таблица 80.1. Векторы, уровни прерываний и устройства в системе Windows

Ведущий контроллер

Номер входа	*Номер линии запроса прерывания*	*Аппаратный вектор*	*Программируемый вектор*	*Устройство*
0	IRQ0	50h	08h	Таймер
1	IRQ1	51h	09h	Клавиатура
2	IRQ2	52h	0Ah	Выход ведомого
3	IRQ3	53h	0Bh	COM2
4	IRQ4	54h	0Ch	COM1
5	IRQ5	55h	0Dh	LPT2
6	IRQ6	56h	0Eh	Гибкий диск
7	IRQ7	57h	0Fh	LPT1

Ведомый контроллер

Номер входа	*Номер линии запроса прерывания*	*Аппаратный вектор*	*Программируемый вектор*	*Устройство*
0	IRQ8	58h	70h	CMOS
1	IRQ9	59h	71h	Нестандартная аппаратура
2	IRQ10	5Ah	72h	–
3	IRQ11	5Bh	73h	–
4	IRQ12	5Ch	74h	–
5	IRQ13	5Dh	75h	–
6	IRQ14	5Eh	76h	Жесткий диск
7	IRQ15	5Fh	77h	–

На старых машинах типа XT с одним контроллером прерываний уровень IRQ2 был свободен и мог использоваться при подключении к компьютеру нестандартной аппаратуры, работающей в режиме прерываний. Этому уровню соответствует вектор 0Ah, который обычно и назначался прикладным обработчикам аппаратных прерываний (разумеется, нестандартное устройство можно было подключить и к любому другому свободному уровню).

В современных машинах, использующих два контроллера прерываний, второй, ведомый контроллер подключается как раз ко входу 2 первого (ведущего). Приход аппаратного прерывания на любой вход ведомого контроллера приводит к возникновению сигнала на входе 2 ведущего, который, однако, сопровождается посылкой в процессор не вектора 0Ah (уровень IRQ2), а вектора, который соответствует в ведомом контроллере пришедшему прерыванию (например, 70h для уровня IRQ8).

Линия IRQ2 внешней магистрали и соответствующий контакт разъема расширения подключаются теперь не ко входу 2 ведущего контроллера, который занят, а ко входу 1 ведомого, которому соответствует уровень IRQ9 и вектор 71h реального режима. Для того чтобы обеспечить программную совместимость старых и новых машин, в систему MS-DOS включен обработчик прерывания 71h, в котором вызывается программное прерывание 0Ah. Таким образом, хотя внешнее устройство, подключенное к

линии IRQ2, фактически работает на уровне IRQ9, его могут обслуживать обработчики прерываний, вызываемые через вектор 0Ah.

В системе Windows перенаправление вектора 71h не реализуется. Поэтому внешние устройства, подключаемые к линии IRQ2 (а фактически к уровню IRQ9, т. е. к входу 1 ведомого контроллера прерываний), необходимо программировать через вектор 71h, соответствующий этому уровню. Устройства же, подключаемые к другим свободным уровням, например к IRQ3 или IRQ5, программируются, как и раньше, через векторы 0Bh или 0Dh. При этом, хотя на первый взгляд вектор 0Dh совпадает по номеру с вектором исключения общей защиты, никаких конфликтов не возникает, так как обработчик исключения общей защиты вызывается через аппаратный вектор 0Dh таблицы IDT, а "вектор 0Dh", используемый в программе, в аппаратном плане соответствует дескриптору IDT с номером 55h, а в программном – ячейке с номером 0Dh в таблице векторов защищенного режима, не имеющей никакого отношения к обработке исключений.

Общие правила обработки прерываний от нестандартной аппаратуры в системах DOS и Windows одинаковы. В основной программе выполняется сохранение исходного содержимого используемого вектора и занесение в вектор полного адреса нового обработчика. Далее необходимо размаскировать используемый уровень прерываний в контроллере прерываний (порт 21h в ведущем контроллере или A1h в ведомом) и, возможно, разрешить прерывания в подключенной к компьютеру аппаратуре, если в регистре управления интерфейсной платы, осуществляющей связь с нестандартной аппаратурой, имеется бит управления прерываниями. Выполнив эти инициализирующие действия, основная программа может заниматься чем угодно. Сигнал прерывания из аппаратуры передаст управление установленному обработчику, который должен перед своим завершением послать в контроллер прерываний команду EOI, чтобы снять блокировку в контроллере прерываний (см. статью 26). Перед завершением всей программы необходимо в общем случае восстановить в используемом векторе его исходное содержимое, замаскировать используемый уровень в контроллере прерываний и запретить прерывания в регистре управления аппаратурой.

Для изучения базовых, архитектурных вопросов обработки прерываний можно воспользоваться простыми обработчиками прерываний от системного таймера, клавиатуры или мыши. Эти периферийные устройства имеются на любом компьютере и везде работают одинаково. Однако обсуждение особенностей обработки прерываний от нестандартной аппаратуры в системе Windows требует привязки к конкретной измерительной или управляющей аппаратуре, которая, разумеется, отсутствует на компьютере пользователя. Приводимые ниже примеры отлаживались на специально разработанном макете, оформленном в виде интерфейсной платы и представляющем собой программируемый таймер-счетчик внешних событий. Макет используется в лабораторном студенческом практикуме по курсам, посвященным программированию аппаратуры физического эксперимента. В частности, с помощью этой интерфейсной платы демонстрируется методика разработки драйверов Windows для управления нестандартной аппаратурой.

Хотя разработанный нами макет является сугубо специфическим устройством, отсутствующим где-либо, кроме лаборатории авторов, в нем используются общепринятые методы построения микропроцессорной аппаратуры, и в этом плане он вместе с компьютером представляет собой типичную программно-управляемую измеритель-

ную установку. Читателю не составит труда трансформировать описываемые ниже примеры применительно к собственным потребностям.

На рис. 80.1 приведена условная логическая схема интерфейсной платы таймера-счетчика, по которой можно проследить ее функционирование и принципы программного управления.

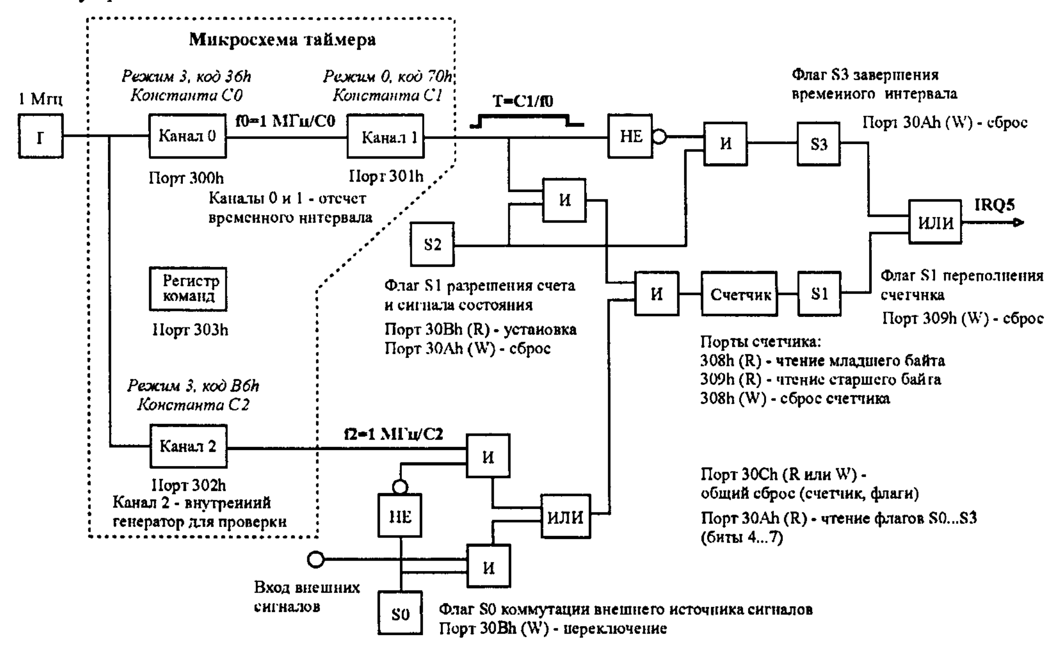


Рис. 80.1. Логическая схема интерфейсной платы таймера-счетчика

Плата предназначена для построения на базе персонального компьютера автоматизированного устройства для счета событий. Плата включает микросхему трехканального таймера КР580ВИ53, 16-битовый счетчик для счета поступающих событий и ряд логических узлов. Счетчик может считать внешние события или импульсы от встроенного в плату генератора. Последний режим используется для проверки работоспособности платы и отладки разрабатываемого программного обеспечения.

Микросхему КР580ВИ53 можно считать прототипом системного таймера, принципы программирования которого, а также возможные режимы были описаны в статье 27. Однако в рассматриваемом устройстве каналы микросхемы используются отлично от таймера компьютера.

Каналы 0 и 1 микросхемы таймера, соединенные на плате последовательно, позволяют задать временной интервал счета событий. Длительность интервала определяется константами C0 и C1, занесенными в буферные регистры каналов 0 и 1, и равна

T = (C0 * C1)/f,

где f – частота импульсов, подаваемых на вход канала 0. Канал 0 работает в режиме 3 пересчета импульсов; канал 1 работает в режиме 0 генерации единственного импульса с длительностью T. На плате установлен кварцевый генератор с частотой 1 МГц, поэтому

T = (C0 * C1) мкс.

Соотношение констант C0 и C1 в принципе не имеет значения, однако в силу некоторой особенности микросхемы таймера (в котором, строго говоря, не предусмотрена возможность последовательного соединения каналов) он дает погрешность приблизительно в ½ периода канала 0, и для ее уменьшения желательно выбирать C0 как можно меньше (однако запрещается значение C0=3).

Канал 2 микросхемы таймера, используемый при отладке программ управления в качестве внутреннего источника считаемых событий, работает в режиме 3 пересчета импульсов. При занесении в его буферный регистр константы C2 частота импульсов на его выходе составляет

$f2 = 10^6/C2$ Гц.

В состав электронной схемы платы входят 4 1-битовых флага S0, S1, S2 и S3, служащие для управления устройством и индикации его состояния. Флаги S0 и S2 являются управляющими, S1 и S3 – индикаторными. Флаг S0 переключает вход счетчика (счет внешних сигналов или импульсов от встроенного генератора), флаг S2 разрешает счет, а также установку флага S3 по завершении временного интервала. Индикаторные флаги указывают: S3 – на завершение установленного временного интервала, S1 – на переполнение счетчика. Установка любого из этих флагов приводит к возбуждению сигнала прерывания на линии IRQ5.

Все 4 флага: S0, S1, S2 и S3 – входят в состав четырехбитового регистра с адресом 30Ah, где они занимают соответственно биты 4, 5, 6 и 7 и могут быть прочитаны командой in ввода из порта. Однако выполнить запись в любой из флагов через порт 30Ah нельзя; для установки и сброса флагов предусмотрены специальные операции. Так, для установки флага S2 разрешения счета следует выполнить чтение из порта 30Bh; для сброса флага S1 переполнения счетчика – запись (чего угодно) в порт 309h. Полный состав операций, реализованных в рассматриваемой схеме, приведен на рис. 80.1.

Общая структурная схема программы управления платой в режиме ожидания завершения временного интервала выглядит следующим образом:

- общий сброс (чтение или запись через порт 30Ch);
- засылка управляющих слов в регистр команд (порт 303h) для задания режима каналов 0, 1 и 2;
- засылка констант счета в буферные регистры каналов 0, 1 и 2 (порты 300h, 301h и 302h);
- установка флага S2 для разрешения счета (чтение из порта 30Bh);
- ожидание установки флага S3, т. е. окончания заданного временного интервала;
- после установки флага S3 считывание содержимого счетчика платы (в два этапа – сначала 1 байт счетчика, затем другой; чтение младшего байта осуществляется через порт 308h, чтение старшего – через порт 309h).

При использовании режима прерываний на этапе инициализации следует выполнить первые 4 из перечисленных выше операций; флаг разрешения-запрещения прерываний в схеме не предусмотрен. Поскольку прерывание возбуждается как при завершении временного интервала, так и при переполнении счетчика, в обработчике прерываний в принципе следует проверять состояние флагов S1 и S3 и определять причину прерывания.

Рассмотрим сначала обычное 16-разрядное приложение Windows, выполняющее управление платой таймера-счетчика в режиме прерываний без всяких драйверов. Что-

Чтобы продемонстрировать проблемы, возникающие при обработке прерываний в приложениях Windows, нам придется сделать эту программу более типичной (ввести в нее главное окно приложения с меню) и, следовательно, заметно более сложной по сравнению с примером 76.1.

Для того чтобы в главном окне приложения появилась стандартная полоска с пунктами меню, необходимо в тексте программы заказать вывод окна с меню, а состав меню описать в специальном файле ресурсов. Этот файл является текстовым файлом и имеет стандартное расширение .RC. Различные ресурсы описываются в нем в специальных форматах, понятных для компилятора ресурсов, входящего в состав интегрированной среды Borland C++. Помимо меню, в файле ресурсов можно описывать состав диалоговых окон, таблицы символьных строк и другие объекты, выводимые в окно приложения. Для того чтобы IDE выполнила обработку файла ресурсов, его необходимо включить в состав проекта, установив флажок на кнопке .rc вкладки Advanced Options панели Target Expert (см. рис. 76.4). Ниже приведен использованный в примере 80.1 файл ресурсов.

Пример 80.1. Управление платой счетчика-таймера в режиме прерываний. Исходный текст файла ресурсов 80-01.RC

```
#define MI_START 100        //Константы, используемые
#define MI_EXIT 101         //для идентификации
#define MI_READ 102         //пунктов меню
Main MENU{
  POPUP "Режим"  {
    MENUITEM "Пуск",MI_START
    MENUITEM "Чтение",MI_READ
    MENUITEM SEPARATOR
    MENUITEM "Выход",MI_EXIT
  }
}
```

Состав, или сценарий, меню описывается в файле ресурсов с помощью предусмотренных для этого ключевых слов. Сценарий меню начинается с ключевого слова MENU, которое может быть написано как прописными, так и строчными буквами. За ним следует перечень пунктов меню, заключенный в фигурные скобки. Слово MENU предваряется произвольным именем (у нас это имя Main), которое выступает как идентификатор меню и будет использовано далее в тексте программы.

Каждый пункт перечня пунктов меню начинается с ключевого слова POPUP, за которым следует название пункта меню. В нашем примере все меню состоит из единственного пункта "Режим".

Вслед за предложением POPUP идет перечень подпунктов этого пункта меню, который появится на экране при его выборе (щелчком левой клавиши мыши). Часто подпункты меню называют просто пунктами или командами меню. Перечень команд заключается в фигурные скобки. Каждая команда начинается ключевым словом MENUITEM, за которым указывается название данной команды и ее идентификатор. Идентификаторы обычно имеют символическую форму. У нас это обозначения MI_START, MI_EXIT и MI_READ, которым в начале файла ресурсов присвоены (оператором препроцессора #define) произвольные значения 100, 101 и 102.

Заметьте, что идентификатор (в виде имени Main) имеет наряду с командами меню также и все меню, в то время как пункты меню, в нашем случае пункт "Режим", идентификатора не имеют. Это происходит потому, что пункты меню обслуживаются сис-

темой Windows, а не прикладной программой. Мы не выполняем никаких действий при выборе пункта "Режим"; система сама открывает перечень подпунктов-команд. Команды же меню обслуживаются приложением, и, чтобы их можно было различить, им присваиваются идентификаторы.

Предложение

```
MENUITEM SEPARATOR
```

служит для проведения в меню горизонтальной черты, разделяющей группы команд и, естественно, идентификатора не имеет.

Перейдем теперь к рассмотрению основного файла проекта примера 80.1 – исходного текста приложения Windows 80-01.CPP.

Пример 80.1 (продолжение). Управление платой счетчика-таймера в режиме прерываний. Исходный текст программного файла приложения Windows 80-01.CPP

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <string.h>
#include <dos.h>              //Поддержка средств MS-DOS
//Определения констант
#define MI_START 100          //Константы, используемые
#define MI_EXIT 101           //для идентификации
#define MI_READ 102           //команд меню
#define VECTOR 13        //Номер используемого вектора прерываний (уровень IRQ5)
//Прототипы функций
void Register(HINSTANCE);  //Вспомогательные
void Create(HINSTANCE);    //функции
void interrupt isr(...);   //Обработчик прерываний
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);//Оконная функция
void OnCommand(HWND,int,HWND,UINT);//Функции обработки
void OnDestroy(HWND);         //сообщений Windows
void InitCard();              //Функция инициализации платы
//Глобальные переменные
void interrupt (*old_isr)(...);//Для сохранения исходного содержимого вектора
char szClassName[]="MainWindow";//Имя класса
char szTitle[]="Управление";//Заголовок окна
int unsigned data,parm;     //Результат измерений, передаваемый в драйвер параметр
unsigned char mask,half;    //Маска и байт для получения результата
//Главная функция WinMain
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE,LPSTR,int){
  MSG msg;                    //Структура типа MSG
  Register(hInstance);        //Зарегистрируем класс главного окна
  Create(hInstance);          //Создадим и покажем окно
  while(GetMessage(&msg,NULL,0,0))//Цикл обработки сообщений
     DispatchMessage(&msg);
  return 0;                   //Выход из приложения в Windows
  }
//Функция Register регистрации класса окна
void Register(HINSTANCE hInst){
  WNDCLASS wc;                //Структура с характеристиками класса окна
  memset(&wc,0,sizeof(wc));//Обнуление структуры wc
  wc.lpszClassName=szClassName;//Имя класса окна
  wc.hInstance=hInst;         //Дескриптор приложения
  wc.lpfnWndProc=WndProc;  //Имя оконной функции
  wc.lpszMenuName="Main";  //Имя меню в файле ресурсов
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);//Дескриптор курсора
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);//Дескриптор пиктограммы
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);//Дескриптор фона окна
```

```
  RegisterClass(&wc);      //Регистрация класса главного окна
  }
//Функция Create создания и показа окна
void Create(HINSTANCE hInst){
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
      10,10,200,100,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  }
//Оконная функция WndProc главного окна
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
  default:
    return(DefWindowProc(hwnd,msg,wParam,lParam));
  }
}
//Функция OnCommand обработки сообщений WM_COMMAND от пунктов меню
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case MI_START:          //Выбрана команда "Пуск"
      old_isr=getvect(VECTOR);//Сохраним исходный вектор
      setvect(VECTOR,isr);//Установим наш обработчик isr
      InitCard();           //Инициализируем плату
      break;
    case MI_READ:           //Выбрана команда "Чтение"
      char txt [80];
      wsprintf(txt,"Накоплено %d событий",data);
      MessageBox(hMainWnd,txt,"Данные",MB_ICONINFORMATION);
      break;
    case MI_EXIT:           //Выбрана команда "Выход"
      DestroyWindow(hwnd); //Завершение приложения
  }
}
//Функция OnDestroy обработки сообщения WM_DESTROY
void OnDestroy(HWND){
  mask|=0x20;               //Установим в маске бит 5
  outportb(0x21,mask);      //И выведем в порт
  setvect(VECTOR,*old_isr);//Восстановим исходный вектор
  PostQuitMessage(0);       //Завершим приложение
}
//Функция InitCard инициализации платы
void InitCard (){
  inportb(0x30c);           //Сброс
//Установим режимы каналов
  outportb(0x303,0x36);     //Канал 0, режим 3
  outportb(0x303,0x70);     //Канал 1, режим 0
  outportb(0x303,0xb6);     //Канал 2, режим 3
//Занесем константы каналов
  parm=100;
  outportb(0x300,LOBYTE(parm));//Младший байт константы C0
  outportb(0x300,HIBYTE(parm));//Старший байт константы C0
  parm=20000;
  outportb(0x301,LOBYTE(parm));//Младший байт константы C1
  outportb(0x301,HIBYTE(parm));//Старший байт константы C1
  parm=1000;
  outportb(0x302,LOBYTE(parm));//Младший байт константы C2
  outportb(0x302,HIBYTE(parm));//Старший байт константы C2
//Размаскируем уровень 5 в контроллере прерываний
  mask=inportb(0x21);      //Прочитаем текущую маску
  mask&=0xdf;               //Сбросим в ней бит 5
```

```
  outportb(0x21,mask);      //Выведем в порт
  inportb(0x30b);           //Пуск
  }
//Функция isr - обработчик прерываний от платы
void interrupt isr(...){
  half=inportb(0x309);      //Чтение старшего байта результата
  data=(WORD)half;
  data<<=8;                 //Сдвиг в старшую половину ячейки
  half=inportb(0x308);      //Чтение младшего байта результата
  data+=half;               //Объединение его со старшим байтом
  outportb(0x20,0x20);      //Команда конца прерываний EOI в контроллер прерываний
  }
```

Как и в случае примера 76.1, программа начинается с группы операторов препроцессора (#include для подсоединения к тексту программы необходимых заголовочных файлов и #define для определения констант), за которой следуют определения прототипов всех используемых в программе функций и описания глобальных переменных.

В типичном приложении Windows главная функция WinMain() должна выполнить по меньшей мере три важных процедуры:

- зарегистрировать в системе Windows класс главного окна. Если помимо главного окна планируется выводить на экран внутренние, порожденные окна, то их классы тоже необходимо зарегистрировать. Windows выводит на экран и обслуживает только зарегистрированные окна;
- создать главное окно и показать его на экране. Порожденные окна также необходимо создать, хотя это можно сделать и позже и не обязательно в функции WinMain();
- организовать цикл обработки сообщений, поступающих в приложение. Вся дальнейшая жизнь приложения будет фактически состоять в бесконечном выполнении этого цикла и в обработке поступающих в приложение сообщений. Запущенное приложение Windows обычно функционирует до тех пор, пока пользователь не подаст команду его завершения с помощью системного меню или вводом Alt+F4. Эти действия приводят к завершению главной функции и удалению приложения из списка действующих задач.

Первое из перечисленных действий выполняется в нашем примере вызовом прикладной функции Register(); второе – вызовом функции (тоже прикладной) Create(); третье – циклом, включающим две функции Windows – GetMessage() и DispatchMessage().

Как уже отмечалось в статье 76, при запуске приложения Windows управление всегда передается функции WinMain(). Эта функция, имея в принципе циклический характер, выполняется в течение всей жизни приложения. Основное назначение функции WinMain() – выполнение инициализирующих действий и организация цикла обработки сообщений.

Сообщения Windows являются, пожалуй, самой важной концепцией этой системы. Каждый раз, когда происходит какое-то событие, затрагивающее интересы программы (например, пользователь выбирает пункт меню или нажимает на кнопку в окне диалога), Windows посылает приложению сообщение об этом событии. Задача функции WinMain() заключается в приеме этого сообщения и передаче его второму важнейшему компоненту любого приложения Windows – функции главного окна, или, проще, оконной функции (оконной процедуре). В отличие от главной функции WinMain(), имя которой изменять нельзя, имя оконной функции может быть любым. В наших

примерах оконная функция главного окна будет именоваться WndProc (от Windows Procedure).

Оконная функция содержит столько фрагментов, сколько конкретных сообщений предполагается обрабатывать в данном приложении. Если, например, приложение должно отслеживать координаты мыши, то в оконную функцию следует включить фрагмент обработки сообщений о перемещении мыши; если, кроме этого, приложение должно реагировать на нажатие клавиш клавиатуры, в оконную функцию включается фрагмент обработки сообщений о нажатии клавиш.

Две остальные функции, входящие в состав приложения, – Register() и Create() введены в программу просто для повышения ее структурированности. Они являются подпрограммами, явно вызываемыми из функции WinMain(). Другими словами, в эти функции вынесены некоторые действия, которые могли бы быть выполнены непосредственно в функции WinMain().

Главная функция приложения WinMain начинается с объявления структурной переменной msg. Это важнейшая переменная, с помощью которой в программу передается содержимое сообщений Windows. Каждое сообщение представляет собой пакет из шести данных, описанных в файле windows.h с помощью структуры типа MSG, состоящей из следующих элементов:

```
struct MSG {
  HWND   hwnd;          //Дескриптор окна, которому адресовано сообщение
  UINT   message;       //Код сообщения
  WPARAM wParam;        //Дополнительная информация (слово)
  LPARAM lParam;        //Дополнительная информация (двойное слово)
  DWORD  time;          //Время отправления сообщения
  POINT  pt;            //Позиция курсора мыши на момент отправления сообщения
  }
```

Сообщения являются реакцией системы Windows на различные происходящие в системе события: движение мыши, нажатие клавиши, срабатывание таймера и т. д. Отличительным признаком сообщения является его код, который может принимать значения (для системных сообщений) от 1 до 0x3FF. Каждому коду соответствует своя символическая константа, имя которой достаточно ясно говорит об источнике сообщения. Так, при движении мыши возникают сообщения WM_MOUSEMOVE (код 0x200), при нажатии на левую клавишу мыши – сообщение WM_LBUTTONDOWN (код 0x201), при срабатывании таймера – WM_TIMER (код 0x113). Сообщения, идущие от пунктов меню, имеют код WM_COMMAND (код 0x111).

Переменная msg будет использована в дальнейшем при вызове функций GetMessage() и DispatchMessage().

Действия по регистрации класса окна мы вынесли для наглядности в отдельную функцию Register(). В ней объявляется и заполняется структура типа WNDCLASS, служащая для описания характеристик класса регистрируемого окна, а затем вызывается функция Windows RegisterClass(), которая и выполняет регистрацию данного класса.

Структура WNDCLASS, определенная в файле windows.h, содержит ряд членов, задающих наиболее общие характеристики окна:

```
struct WNDCLASS {
    UINT       style;          //Стиль окна
    WNDPROC    lpfnWndProc;    //Имя оконной функции
    int        cbClsExtra;     //Число байтов дополнительной информации о классе
```

```
int       cbWndExtra;     //Число байтов дополнительной информации об окне
HINSTANCE hInstance;      //Дескриптор приложения
HICON     hIcon;          //Дескриптор пиктограммы приложения
HCURSOR   hCursor;        //Дескриптор курсора приложения
HBRUSH    hbrBackground;  //Дескриптор кисти для фона окна
LPCSTR    lpszMenuName;   //Указатель на строку с именем меню окна
LPCSTR    lpszClassName;  //Указатель на строку с именем класса окна
}
```

В такой простой программе, как наша, нет необходимости определять все члены этой структуры; для упрощения дела мы сначала обнуляем всю структуру командой

```
memset(&wc,0,sizeof(wc)
```

которая записывает, начиная с адреса &wc, нули в количестве sizeof(wc) штук, а затем задаем значения только интересующим нас членам.

Наиболее важными для дальнейшего функционирования программы являются три поля: hInstance, где хранится дескриптор данного приложения; lpszClassName, куда заносится произвольное имя, присвоенное нами окнам данного класса (у нас это окно единственное) и lpfnWndProc – имя оконной функции. Именно с помощью структуры WNDCLASS Windows, обслуживая нашу программу, определяет адрес оконной функции, которую она должна вызывать при поступлении в окно сообщений. Поля lpszClassName и lpszClassName мы заполняем из глобальных переменных, а значение дескриптора приложения поступает в функцию Register() через ее аргумент hInst.

Еще один важный для нашей конкретной программы элемент структуры WNDCLASS – поле lpszMenuName, в которое записывается имя меню в файле ресурсов. Именно заполнение этого элемента определяет наличие в главном окне приложения (когда оно будет создано и выведено на экран) полоски с пунктами меню, как мы их задали в сценарии меню.

Менее важными в принципиальном плане, но существенными для разумного поведения приложения являются поля hIcon, hCursor и hbrBackground, куда следует записать дескрипторы пиктограммы приложения, его курсора и цвета фона (точнее, цвета кисти, которая используется для закраски фоновой области окна).

Пиктограмма и курсор относятся к ресурсам Windows, которые обычно загружаются из специально созданного файла ресурсов с помощью функций Windows LoadCursor() и LoadIcon(). В качестве первого аргумента этих функций указывается дескриптор приложения, в котором хранится требуемый ресурс, а в качестве второго – имя ресурса. Однако можно обойтись и встроенными ресурсами Windows. Для этого в качестве первого аргумента этих функций указывается NULL (обычно NULL на месте дескриптора приложения обозначает саму систему Windows); второй аргумент надо выбрать из списка встроенных ресурсов Windows. Получить списки встроенных ресурсов можно с помощью интерактивного справочника IDE Borland C++ (статьи LoadCursor и LoadIcon).

Зарегистрировав класс окна, можно приступить к его созданию и показу. Эта процедура вынесена у нас в подпрограмму Create(), в которой последовательно вызываются две функции Windows: CreateWindow() – для создания главного окна и ShowWindow() – для его вывода на экран.

Функция CreateWindow() имеет 11 параметров. На первом месте указывается класс создаваемого окна, на втором – его заголовок. Далее следует константа, описывающая

стиль данного окна. Использованная в нашей программе константа WS_OVERLAPPEDWINDOW типична для главных окон приложений – задаваемый ею стиль создает окно с толстой рамкой и стандартной полоской в верхней части окна, включающей кнопку системного меню, заголовок и три кнопки для управления размерами окна (см., напр., рис. 79.1). Вслед за константой стиля указываются координаты верхнего левого угла окна и его размеры. Последующие параметры для нас не представляют интереса.

Функция CreateWindow() возвращает (при успешном выполнении) дескриптор созданного окна. Во многих случаях этот дескриптор используется при последующих обращениях к данному окну, и его полезно сохранить в глобальной переменной. В данном случае дескриптор окна нужен только для передачи его в функцию ShowWindow(), поэтому он сохраняется в локальной переменной hwnd, существующей лишь пока выполняется подпрограмма Create().

После выполнения функции ShowWindow() на экране появляется главное окно с заданными нами характеристиками. Теперь для правильного функционирования приложения необходимо организовать цикл обработки сообщений.

Цикл обработки сообщений в простейшем виде состоит из одного предложения языка Си:

```
while(GetMessage(&msg,NULL,0,0))
    DispatchMessage(&msg);
```

В этом бесконечном (если его не разорвать изнутри) цикле вызывается функция Windows GetMessage() и, если она возвращает ненулевое значение, вызывается функция DispatchMessage().

Функция GetMessage() анализирует очередь сообщений приложения. Если в очереди обнаруживается сообщение, GetMessage() изымает его из очереди и передает в структуру msg, после чего завершается с возвратом значения TRUE. Если сообщений в очереди нет, функция GetMessage() вызывает программы Windows, которые передают управление другим работающим приложениям (их циклам обработки сообщений). После опроса остальных приложений управление возвращается в наше приложение в ту же точку анализа очереди сообщений. Таким образом, функция GetMessage() завершится (с возвратом значения TRUE) лишь после того, как очередное сообщение попадет в структуру msg.

Далее в цикле while вызывается функция DispatchMessage(). Ее назначение – вызов оконной функции для того окна, которому предназначено очередное сообщение. После того как оконная функция обработает сообщение, возврат из нее приводит к возврату из функции DispatchMessage() на продолжение цикла while.

Функция GetMessage() требует 4 параметра. Первый из них – адрес структуры msg, в которую GetMessage() должна передать изъятое из очереди сообщение. Второй параметр типа HWND позволяет определить окно, чьи сообщения будут изыматься функцией GetMessage(). Если этот параметр равен NULL, GetMessage() работает со всеми сообщениями данного приложения.

Два последних параметра определяют диапазон сообщений, которые анализируются функцией GetMessage(). Если, например, в качестве этих параметров указать константы WM_KEYFIRST и WM_KEYLAST, GetMessage будет забирать из очереди

только сообщения, относящиеся к клавиатуре; константы WM_MOUSEFIRST и WM_MOUSELAST позволят работать только с сообщениями от мыши. Чаще всего надо анализировать все сообщения. Чтобы исключить фильтрацию сообщений, оба параметра должны быть равны нулю.

Особая ситуация возникает, если функция GctMessage() обнаруживает в очереди сообщение WM_QUIT с кодом 0x12. В этом случае GetMessage() сразу же завершается с возвратом значения FALSE. Однако цикл while выполняется, лишь если GetMessage() возвращает TRUE. Возврат FALSE приводит к завершению цикла и переходу на предложение

```
return 0;
```

т. е. к завершению функции WinMain() и всего приложения.

Однако откуда берется сообщение WM_QUIT? Обычно приложения Windows завершают свою работу по команде пользователя. Он может щелкнуть по кнопке завершения (с крестиком) в правом верхнем углу окна, дважды щелкнуть по маленькой пиктограмме в левом верхнем углу окна, вызвать системное меню одиночным щелчком по этой пиктограмме и выбрать пункт "Закрыть" или, наконец, нажать клавиши Alt+F4. Во всех этих случаях Windows убирает с экрана окно приложения и посылает в приложение сообщение WM_DESTROY.

В чем должна состоять обработка этого сообщения? В процессе своего выполнения программа приложения могла использовать те или иные ресурсы Windows: создать кисти, перья или шрифты, установить таймеры, динамически выделить память и т. д. Перед завершением приложения эти ресурсы следует освободить, иначе можно вывести из строя всю систему. Возможно также, что программа использовала какие-то средства (не связанные с Windows), которые перед завершением работы следует привести в порядок: закрыть открытые файлы, выключить включенную аппаратуру и т. д. Наконец, можно вывести на экран предупреждающее сообщение.

Выполнив все эти завершающие действия, программа должна вызвать функцию Windows PostQuitMessage(). Эта функция как раз и генерирует сообщение Windows WM_QUIT, которое, как уже отмечалось выше, приводит к разрыву цикла while обработки сообщений, выполнения последнего оператора return функции WinMain и завершению программы.

Подведем итоги изложенного. Типичное приложение Windows имеет стандартную структуру, изображенную на рис. 80.2. В программе можно выделить два основных элемента: главную функцию и оконную функцию главного окна. В главной функции прежде всего осуществляются подготовительные действия: регистрация в Windows класса главного окна, а также создание и показ этого окна. Затем главная функция входит в бесконечный цикл обработки сообщений. Каждое поступившее в окно сообщение вызывает к жизни оконную функцию.

Главная функция WinMain()

1. Регистрация класса окна
2. Создание и показ окна
3. Цикл обработки сообщений

Оконная процедура

Если WM_TIMER, то ...
Если WM_COMMAND, то ...
Если WM_DESTROY, то PostQuitMessage()
Если любое ненужное сообщение, то DefWindowProc()

Рис. 80.2. Структура типичного приложения Windows

Оконная функция главного окна содержит столько фрагментов, сколько предполагается обрабатывать сообщений. Конкретное содержимое каждого фрагмента определяет программист. Все сообщения, не нуждающиеся в прикладной обработке, должны поступать в функцию обработки сообщений по умолчанию DefWindowProc() и обрабатываться системой Windows. Эти действия можно описать в программе с помощью конструкции switch-case:

```
switch(msg){
  case WM_COMMAND:
//Действия в ответ на выбор пункта меню
  return 0;
  case WM_DESTROY:
    //Действия, необходимые перед завершением программы
    PostQuitMessage(0);//Завершение программы
    return 0;
  default:
    return(DefWindowProc(hwnd,msg,wParam,lParam));
  }
```

Однако такая конструкция неудобна. Оконная функция представляет собой в этом случае один длинный оператор switch со столькими блоками case, сколько сообщений Windows предполагается обрабатывать в программе. При обработке ряда сообщений, например WM_COMMAND, внутрь блоков case приходится включать вложенные операторы switch-case, да еще не одного уровня вложенности. В результате функция WndProc() становится чрезвычайно длинной и запутанной. Весьма полезная идея структурированности программы исчезает почти полностью, так как все приложение оказывается состоящим из сдва ли не единственной функции WndProc() со множеством развствлений внутри.

Заметного упрощения структуры программы можно добиться, используя макрос HANDLE_MSG, определенный в файле windowsx.h. Этот макрос, в зависимости от его параметров, преобразуется в один из многочисленных макросов вида HANDLE_WM_COMMAND, HANDLE_WM_DESTROY и т. д., так же определенных в файле windowsx.h. При использовании этих макросов все процедуры обработки сообщений выделяются в отдельные функции, а в оконной функции WndProc() остаются только строки переключения на эти функции в случае прихода того или иного сообщения. Оконная функция, даже при большом количестве обрабатываемых сообщений, становится короткой и чрезвычайно наглядной; наличие же для обработки каждого сообщения отдельной функции также весьма упрощает разработку их алгоритмов, и особенно отладку.

В программе примера 80.1 обрабатываются всего два сообщения: WM_COMMAND и WM_DESTROY. Как уже говорилось выше, сообщение WM_COMMAND поступает в главное окно приложения в случае выбора пользователем какой-либо команды (подпункта) меню. В составе пакета данных этого сообщения из Windows в приложение приходит значение идентификатора выбранной команды. Сообщение WM_DESTROY поступает в приложение, если пользователь выполнил действия, приводящие к завершению работы приложения, – нажал крестик в правом верхнем углу главного окна или ввел с клавиатуры команду Ctrl+F4.

Соответственно в программу введены две функции обработки этих сообщений OnCommand() и OnDestroy(). Формат этих функций, т. е. количество параметров и их типы, а также тип возвращаемого значения можно найти в файле windowsx.h. Любопытно отметить, что большая часть функций обработки сообщений ничего не возвращает и не требует оператора return, потому что он автоматически включается в текст программы при расширении макроса HANDLE_MSG.

При запуске программы примера 80.1 на экран будет выведено главное окно и программа будет ждать команд оператора. Вывод программы с развернутым меню "Режим", приведен на рис. 80.3.

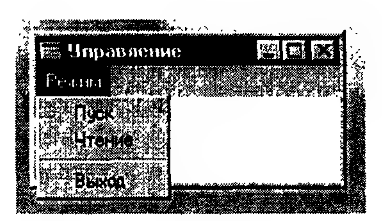


Рис. 80.3. Главное окно программы

Для обеспечения работы системы прерываний в начале программы указан прототип функции isr() с описателем interrupt, которая будет служить в качестве прикладного обработчика прерываний, а в области глобальных переменных объявлен указатель old_isr на функцию типа interrupt, где будет храниться адрес исходного обработчика.

При выборе команды "Пуск" (идентификатор MI_START) с помощью функции Си getvector() читается и сохраняется в переменной old_isr исходное содержимое вектора 13, после чего функцией setvector() в вектор 13 заносится адрес нашего обработчика isr. Как было показано в начале этой статьи, функции getvector() и setvector() работают не с истинной таблицей дескрипторов прерываний IDT, а со специальной таблицей векторов защищенного режима, входящей в состав виртуальной машины.

Процедура инициализации платы вынесена в отдельную функцию InitCard(). В ней прежде всего выполняется общий сброс всех узлов платы, далее посылкой кодов 36h, 70h и B6h в порт регистра команд 303h задаются режимы всех трех каналов таймера, и, наконец, во все три канала последовательно засылаются константы C0, C1 и C2. Для простоты значения констант определены прямо в тексте программы; в реальной программе их следовало бы вводить с клавиатуры. Для выделения младших и старших половин констант (вспомним, что константы размером в слово засылаются в порты таймера в два приема, по байтам) используются удобные макросы Windows LOBYTE и HIBYTE.

Нам осталось размаскировать уровень 5 (соответствующий вектору 13) контроллера прерываний. Текущая маска читается в переменную mask, оператором & в ней обнуляется бит 5 (нулю в бите 5 и единице в остальных битах байта соответствует число 0DFh), и новое значение маски отправляется в порт 21h. Еще раз заметим, что VMM перехватывает обращение к запрещенному порту 21h, в результате чего наше данное поступает не в аппаратный порт контроллера прерываний, а в его программную ко-

пию – виртуальную маску прерываний, входящую в состав виртуальной машины. "Заведует" этой маской, как и всей системой прерываний, виртуальный контроллер прерываний VPICD.

Последним предложением функции InitCard() в схеме счетчика-таймера устанавливается флаг S2, открывающий вход счетчика и разрешающий установку флага S3 завершения временного интервала.

Обработчик прерываний в реальной установке может выполнять различные функции. Наверное, самое естественное – это вывести на экран сообщение об окончании измерений. Однако сделать это не так-то просто. Дело в том, что возможности обработчика прерываний ограничены. В нем можно выполнять вычислительные действия, а также чтение и запись ячеек памяти, но нельзя, например, вызвать функцию Windows MessageBox() для вывода на экран сообщения. Чтобы не усложнять рассматриваемый пример, мы в функции isr() просто читаем содержимое счетчика платы и заносим его в глобальную переменную data, давая тем самым программе возможность работать с этими данными. Чтение числа накопленных событий приходится выполнять в два приема. Старшая половина данных из порта 309h читается в байтовую переменную half, переносится в переменную data с преобразованием в слово и сдвигается в этой переменной влево на 8 бит, т. е. переносится в старший байт. Далее младшая половина данных читается в ту же переменную half и складывается со старшей в переменной data.

Последнее (обязательное!) действие в обработчике прерываний – снятие блокировки в контроллере прерываний посылкой команды EOI. И эта операция в действительности выполняется не в физическом контроллере прерываний, а в виртуальном.

При выборе пункта меню "Чтение" выполняется относительно простая операция преобразования с помощью функции Windows wsprintf() числа в переменной data в символьную форму и вывод его в окно сообщения функцией MessageBox(). Результат этих действий (для значений констант из примера 80.1) показан на рис. 80.4.

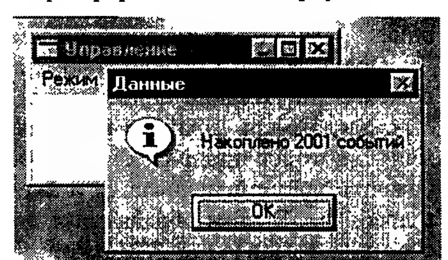


Рис. 80.4. Вывод на экран накопленного числа событий

Таким образом, программное управление аппаратурой, работающей в режиме прерываний, можно реализовать и без использования виртуального драйвера. При этом структуры, макросы и функции языка Си, относящиеся к обслуживанию прерываний (в частности, функции getvect(), setvect(), inport(), outport() и др.), описаны в заголовочном файле DOS.H, который необходимо подключить к исходному тексту программы. Отметим, что сказанное относится только к 16-разрядным приложениям. Для 32-разрядных приложений аналога файла DOS.H нет, и обработка прерываний в таких программах невозможна без соответствующего виртуального драйвера. Этот вопрос будет рассмотрен в статье 85. Однако и для 16-разрядных приложений разработка специализированного виртуального драйвера может оказаться полезной, так как при его использовании заметно сокращается время реакции на прерывание, что в некоторых случаях может оказаться существенным для работы аппаратуры.

Принципам построения виртуальных драйверов для обслуживания аппаратных прерываний посвящена следующая статья.

Статья 81. **Виртуальный драйвер для обслуживания аппаратных прерываний**

Как уже отмечалось ранее, виртуальные драйверы служат прежде всего для виртуализации аппаратуры, т. е. для предоставления одновременно выполняемым задачам возможности совместного использования устройств компьютера. Измерительная или управляющая аппаратура, подключаемая к компьютеру с целью создания автоматизированной установки, вряд ли будет эксплуатироваться в многозадачном режиме, однако использование для ее управления виртуального драйвера может заметно сократить программные издержки и уменьшить время отклика. Рассмотрим пример виртуального драйвера, обслуживающего прерывания от интерфейсной платы счетчика-таймера, описанной в предыдущей статье.

Очевидно, что в состав такого драйвера должен входить обработчик прерываний от платы. Функции этого обработчика определяет программист; в простейшем случае обработчик может просто прочитать данное из выходного регистра счетчика и замаскировать прерывания. Однако приложение при этом не узнает о завершении измерений; более естественно организовать вызов из обработчика прерываний драйвера некоторой функции приложения, в которую можно передать прочитанные данные. Фактически эта функция будет играть роль обработчика прерываний приложения, однако вызываться она будет не самим прерыванием, а обработчиком драйвера.

Предусмотрим следующую схему взаимодействия приложения и драйвера. Приложение по команде пользователя вызывает драйвер и передает ему константы настройки таймера. Драйвер инициализирует таймер и возвращает управление в приложение, которое продолжает свое выполнение. По сигналу прерывания от таймера приложение приостанавливается и активизируется обработчик прерываний, находящийся в драйвере. В этом обработчике выполняется чтение данного из выходного регистра счетчика, вызов (через VMM) обработчика прерываний приложения и передача в него считанного данного. Обработчик прерываний приложения принимает из драйвера данное и, завершаясь, возвращает управление в VMM, который, наконец, передает управление в приложение в точку его приостановки.

Поскольку обработчик прерываний приложения вызывается из VMM и после своего завершения опять передает управление в систему, его возможности ограничены. В нем, в частности, недопустим вызов функции MessageBox() вывода информации в окно сообщения. Поэтому здесь, как и в предыдущем примере, возникает проблема оповещения главной функции WinMain() о приходе прерывания. Мы решим эту проблему следующим образом: обработчик прерываний приложения, получив управление, устанавливает флаг, который опрашивается главной функцией в каждом шаге цикла обработки сообщений. Главная функция, обнаружив установленное состояние флага, посылает в приложение сообщение пользователя, которое наравне с сообщениями Windows обслуживается циклом обработки сообщения. В функции обработки этого сообщения уже можно выполнять любые действия, без каких-либо ограничений.

При разработке приложения, реализующего описанный алгоритм (пример 81.1), за основу взят программный комплекс примера 80.1. В проект приложения входят два

файла: файл ресурсов 81-01.RC, в точности совпадающий с файлом 80-01.RC, и программный файл 81-01.CPP, являющийся модификацией файла 80-01.CPP.

Пример 81.1. Обслуживание аппаратных прерываний с помощью виртуального драйвера. Файл 81-01 с исходным текстом приложения Windows

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <string.h>
//Определения констант
#define MI_START 100        //Константы, используемые
#define MI_EXIT 101         //для идентификации
#define MI_READ 102         //пунктов меню
/* void Cls_OnUser(HWND hwnd) */
#define HANDLE_WM_USER(hwnd, wParam, lParam, fn) \//Макрос для обработки
        ((fn)(hwnd), 0L)    // сообщения WM_USER
//Прототипы функций
void Register(HINSTANCE); //Вспомогательные
void Create(HINSTANCE);   //функции
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);//Оконная функция
void OnCommand(HWND,int,HWND,UINT);//Функции
void OnDestroy(HWND);       //обработки сообщений Windows
void OnUser(HWND);          //и пользователя
void InitCard();            //Функция инициализации платы (через драйвер)
void GetAPIEntry();         //Функция получения точки входа в драйвер
void isr(int,int);          //Прототип обработчика прерывания приложения
//Глобальные переменные
char szClassName[]="MainWindow";//Имя класса главного окна
char szTitle[]="Управление";//Заголовок окна
HWND hMainWnd;              //Дескриптор главного окна
FARPROC VxDEntry;           //Адрес точки входа в драйвер
int unsigned data;          //Данное из драйвера
char request;               //Флаг окончания измерений
//Главная функция WinMain
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE,LPSTR,int){
  MSG msg;                  //Структура для приема сообщений
  Register(hInstance);      //Регистрация класса главного окна
  Create(hInstance);        //Создание и показ главного окна
/*Более сложный цикл обработки сообщений, в который включены анализ флага
request завершения измерений и посылка приложению сообщения WM_USER при уста-
новке этого флага*/
  do{
    if(PeekMessage(&msg,NULL,0,0,PM_REMOVE)){
      if (msg.message == WM_QUIT)return msg.wParam;
      DispatchMessage(&msg);
      }//Конец if(PeekMessage())
    if(request){
      request=0;            //Сброс флага
      PostMessage(hMainWnd,WM_USER,0,0);/*Поставить в очередь
           сообщение WM_USER без параметров*/
      }//Конец if(request)
    }while(1);//Конец do
  }//Конец WinMain
//Функция Register регистрации класса окна
void Register(HINSTANCE hInst){
  WNDCLASS wc;
  memset(&wc,0,sizeof(wc));
  wc.lpszClassName=szClassName;
  wc.hInstance=hInst;
  wc.lpfnWndProc=WndProc;
```

```
  wc.lpszMenuName="Main";
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  RegisterClass(&wc);
  }
//Функция Create создания и показа окна
void Create(HINSTANCE hInst){
  hMainWnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
     10,10,200,100,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hMainWnd,SW_SHOWNORMAL);
  }
//Оконная функция WndProc главного окна
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    HANDLE_MSG(hwnd,WM_USER,OnUser);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
//Функция OnCommand обработки сообщений WM_COMMAND от команд меню
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case MI_START://Инициализация платы
      InitCard();
      break;
    case MI_READ:
      char txt [80];
      wsprintf(txt,"Накоплено %d событий",data);
      MessageBox(NULL,txt,"Данные",MB_ICONINFORMATION);
      break;
    case MI_EXIT://Завершить приложение
      DestroyWindow(hwnd);
    }
  }
//Функция OnUser обработки сообщения WM_USER
void OnUser(HWND){
  MessageBox(NULL,"Измерения окончены","Контроль",MB_ICONINFORMATION);
  }
//Функция OnDestroy обработки сообщения WM_DESTROY
void OnDestroy(HWND){
  PostQuitMessage(0);
}
//Функция InitCard инициализации (через драйвер) платы
void InitCard (){
  GetAPIEntry();            //Получение адреса точки входа в драйвер
  _AX=_DS;                  //Передача в драйвер DS
  _BX=55;                   //Канал 0
  _CX=20000;                //Канал 1; BX*CX=Длительность интервала
  _DX=1000;                 //Канал 2, внутренний генератор
  _SI=OFFSETOF(isr);        //Смещение обработчика прерываний
  _DI=SELECTOROF(isr);      //Селектор обработчика прерываний
  VxDEntry();               //Вызов драйвера
  }
//Функция GetAPIEntry получения адреса точки входа в драйвер
void GetAPIEntry(){
  asm{
    mov  AX,0x1684
    mov  BX,0x8000
```

```
    int  0x2F
    mov  word ptr VxDEntry,DI
    mov  word ptr VxDEntry+2,ES
    }
  }
//Обработчик прерываний приложения. Вызывается VMM и возвращает управление в VMM
void isr(int segment,int dt){
  _DS=segment;              //Инициализируем DS селектором, полученным из драйвера
  request++;                //Поставим запрос на сообщение
  data=dt;                  //Получим из драйвера аппаратные данные
  }
```

Главная функция WinMain() выполняет три характерные для нее процедуры: регистрацию класса главного окна, создание и показ главного окна и цикл обработки сообщений. В цикле обработки сообщений имеется принципиальное отличие от примера 80.1: в каждом шаге цикла проверяется состояние флага завершения измерений request и, если флаг оказывается установлен, вызывается функция Windows PostMessage(), которая ставит в очередь сообщений нашего приложения наше же сообщение с кодом WM_USER. Для того чтобы в цикл обработки сообщений включить проверку флага, пришлось заменить в нем функцию GetMessage() на функцию PeekMessage(), которая, в отличие от GetMessage(), при отсутствии сообщений в очереди возвращает управление в программу, что и дает возможность включить в цикл дополнительные действия. Однако PeekMessage() не анализирует сообщение WM_QUIT о завершении программы, поэтому "вылавливание" этого сообщения (и завершение программы оператором return 0 в случае его появления) приходится выполнять вручную. Конструкция

```
do{
...
  }while(1)
```

позволяет организовать бесконечный цикл, поскольку условие продолжения цикла, анализируемое оператором while, безусловно удовлетворяется (константа 1 никогда не может стать равной нулю).

В оконной функции WndProc() фиксируется приход трех сообщений: WM_COMMAND – от пунктов меню, WM_DESTROY – от команд завершения приложения и WM_USER, свидетельствующего об окончании измерений. Поскольку для сообщения WM_USER в файле windowsx.h отсутствует макрос HANDLE_WM_USER, его пришлось определить в начале программы с помощью оператора #define, построив макрорасширение по аналогии с каким-либо из макросов вида HANDLE_*сообщение* из файла windowsx.h, хотя бы макросом HANDLE_WM_DESTROY.

Фрагмент программы, выполняемый при выборе пользователем пункта меню "Пуск", содержит лишь вызов функции InitCard(). В ней вызовом вложенной функции GetAPIEntry определяется адрес API-процедуры драйвера, а затем, после заполнения ряда регистров параметрами, передаваемыми в драйвер, вызывается эта процедура. В драйвер передаются следующие параметры: селектор сегмента данных приложения, три константы для инициализации платы, а также селектор и смещение обработчика прерываний приложения isr(). Передача в драйвер содержимого сегментного регистра DS (селектора сегмента данных) необходима потому, что при вызове драйвером (точнее, VMM) нашей функции isr() не восстанавливается операционная среда приложения, в частности регистр DS не указывает на поля данных приложения, которые оказываются таким образом недоступны. Передав в драйвер содержимое DS, мы сможем

вернуть его назад вместе с другими передаваемыми из драйвера в приложение данными и восстановить тем самым адресуемость данных.

При выборе пользователем пунктов меню "Чтение" или "Выход" выполняются те же действия, что и в примере 80.1.

По сравнению с предыдущим примером упростилась функция OnDestroy(). Поскольку восстановление маски в контроллере прерываний возложено теперь на драйвер, а исходный вектор мы в этом варианте программы не восстанавливаем, в функции OnDestroy() лишь вызывается функция Windows PostQuitMessage(), приводящая к завершению программы.

В обработчике прерываний приложения isr() после засылки в регистр DS нашего же селектора сегмента данных, переданного ранее в драйвер и полученного из него в качестве первого параметра функции isr(), выполняется инкремент флага request и пересылка в переменную data второго параметра функции isr() – результата измерений.

Перейдем к рассмотрению программы виртуального драйвера, входящего в состав нашего программного комплекса (пример 81.1, продолжение).

Пример 81.1 (продолжение). Обслуживание аппаратных прерываний с помощью виртуального драйвера. Файл 81-01.ASM с исходным текстом виртуального драйвера

```
.386p
.XLIST
include vmm.inc
include vpicd.inc
.LIST
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h, \
     Undefined_Init_Order,,API_Handler
;======================
VxD_REAL_INIT_SEG
BeginProc VMyD_Real_Init
        mov   AH,09h
        mov   DX,offset msg
        int   21h
        mov   ax, Device_Load_Ok
        xor   bx, bx
        xor   si, si
        xor   edx, edx
        ret
msg db 'Виртуальный драйвер VMyD загружен'
EndProc   VMyD_Real_Init
VxD_REAL_INIT_ENDS
;======================
VxD_DATA_SEG
Data    dw    0           ;Ячейка для результата измерений
DSseg   dw    0           ;Ячейка для хранения селектора приложения
Segment_Callback dw 0     ;Селектор функции isr приложения
Offset_Callback dd 0      ;Смещение функции isr приложения
IRQ_Handle dd  0          ;Дескриптор виртуального прерывания
VMyD_Int13_Desc label dword;32-битовый адрес следующей далее структуры
VPICD_IRQ_Descriptor <5,,OFFSET32 VMyD_Int_13>;Структура с информацией
                          ;о виртуализованном прерывании
VxD_DATA_ENDS
;======================
VxD_CODE_SEG
BeginProc VMyD_Control
;Включим в состав драйвера процедуру обработки системного сообщения
;Device_Init об инициализации драйвера
Control_Dispatch Device_Init, VMyD_Device_Init
```

```
        clc
        ret
EndProc VMyD_Control
;----------------------
;Процедура, вызываемая при инициализации драйвера системой
BeginProc VMyD_Device_Init
        mov    EDI,OFFSET32 VMyD_Int13_Desc;Адрес структуры
VPICD_IRQ_Descriptor
        VxDCall VPICD_Virtualize_IRQ;Виртуализация устройства
        mov    IRQ_Handle,EAX;Сохраним дескриптор виртуального IRQ
        clc
        ret
EndProc VMyD_Device_Init
;-------------------------
;API-процедура, вызываемая из приложения
;При вызове:
;       AX=DS приложения, BX=C0, CX=C1, DX=C2,
;       DI=селектор isr, SI=смещение isr
BeginProc API_Handler
;Получим параметры из приложения
        push   [EBP.Client_DI]
        pop    Segment_Callback
        push   [EBP.Client_AX];DS
        pop    DSseg
        movzx  ESI,[EBP.Client_SI]
        mov    Offset_Callback,ESI
;Общий сброс
        mov    DX,30Ch
        in     AL,DX
;Размаскируем уровень 5 в физическом контроллере прерываний
        mov    EAX,IRQ_Handle
        VxDCall VPICD_Physically_Unmask
;Засылаем управляющие слова по каналам
        mov    DX,303h
        mov    AL,36h      ;Канал 0
        out    DX,AL
        mov    AL,70h      ;Канал 1
        out    DX,AL
        mov    AL,0B6h     ;Канал 2
        out    DX,AL
;Засылаем константы в каналы
        mov    DX,300h     ;Канал 0
        mov    AX,[EBP.Client_BX];Константа C0
        out    DX,AL       ;Младший байт частоты
        xchg   AL,AH
        out    DX,AL       ;Старший байт частоты
        mov    DX,301h     ;Канал 1
        mov    AX,[EBP.Client_CX];Константа C1
        out    DX,AL       ;Младший байт интервала
        xchg   AL,AH
        out    DX,AL       ;Старший байт интервала
        mov    DX,302h     ;Канал 2
        mov    AX,[EBP.Client_DX];Константа C2
        out    DX,AL       ;Младший байт частоты
        xchg   AH,AL
        out    DX,AL       ;Старший байт частоты
;Установим флаг S2 разрешения счета
        mov    DX,30Bh
        in     AL,DX
        ret
EndProc API_Handler
```

```
;
;Процедура обработки аппаратного прерывания IRQ5 (вектор 13)
BeginProc VMyD_Int_13, High_Freq
        pushad              ;Сохраним все регистры
;Получим результат измерений из выходного регистра счетчика
        mov    DX,309h      ;Порт старшего байта
        in     AL,DX        ;Получим старший байт
        mov    AH,AL        ;Отправим его в AH
        dec    DX           ;DX=308h
        in     AL,DX        ;Получим младший байт
        mov    Data,AX      ;Весь результат в Data
;Выполним завершающие действия в PIC и вызовем функцию приложения
        mov    EAX,IRQ_Handle
        VxDCall VPICD_Phys_EOI;EOI в физический контроллер прерываний
        VxDCall VPICD_Physically_Mask;Маскируем наш уровень
;Перейдем на синхронный уровень
        mov    EDX,0        ;Данные отсутствуют
        mov    ESI,OFFSET32 Reflect_Int;Адрес синхронной процедуры
        VMMCall Call_VM_Event;Установим запрос на ее вызов из VMM
        popad               ;Восстановим все регистры
        clc
        ret
EndProc VMyD_Int_13
;-------------------------
;Процедура уровня отложенных прерываний
BeginProc Reflect_Int
        Push_Client_State ;Выделим место на стеке для регистров клиента
        VMMCall Begin_Nest_Exec;Начнем вложенный блок выполнения
        mov    AX,Data      ;Отправим данное
        VMMCall Simulate_Push;в стек клиента
        mov    AX,DSseg     ;Отправим полученный ранее DS
        VMMCall Simulate_Push;в стек клиента
        mov    CX,Segment_Callback;Зашлем полученный ранее адрес функции isr
        mov    EDX,Offset_Callback;в CS:IP клиента, чтобы после возврата из VMM
        VMMCall Simulate_Far_Call;в виртуальную машину вызвалась эта функция
        VMMCall Resume_Exec;Возврат из VMM в текущую виртуальную машину
        VMMCall End_Nest_Exec;Завершим вложенный блок выполнения
        Pop_Client_State  ;Освободим место на стеке для регистров клиента
        clc
        ret
EndProc Reflect_Int
VxD_CODE_ENDS
end VMyD_Real_Init
```

В сегменте данных драйвера зарезервирован ряд ячеек для временного хранения полученных из приложения параметров, а также результата измерений. Особняком стоит ячейка IRQ_Handle, в которой хранится дескриптор виртуального прерывания. Этот дескриптор назначается системой на этапе инициализации драйвера и остается неизменным все время его жизни, т. е. до перезагрузки компьютера.

Макрос VPICD_IRQ_Descriptor позволяет описать в полях данных структуру с информацией о виртуализованном прерывании. Обязательными элементами этой структуры является номер уровня виртуализуемого прерывания и адрес обработчика аппаратного прерывания (VMyD_Int_13 в нашем случае), включаемый в состав драйвера. Для того чтобы макросы виртуального контроллера прерываний были доступны ассемблеру, к программе необходимо подключить (оператором include) файл VPICD.INC.

Виртуализация прерывания осуществляется на этапе инициализации драйвера. До сих пор мы в явной форме не использовали процедуру VMyD_Control, в которой обрабатыва-

ются системные сообщения Windows. В рассматриваемом драйвере в состав этой процедуры с помощью макроса Control_Dispatch включена процедура VMyD_Device_Init (имя произвольно), которая будет вызвана при получении драйвером системного сообщения Device_Init. Для обработки большего числа сообщений Windows в процедуру VMyD_Control следует включить по макросу Control_Dispatch на каждое обрабатываемое сообщение (с указанием имен сообщения и процедуры его обработки).

Процедура VMyD_Device_Init содержит вызов функции виртуального контроллера прерываний (VPICD) VPICD_Virtualize_IRQ. Эта функция осуществляет виртуализацию указанного уровня прерываний и возвращает дескриптор виртуального прерывания, который сохраняется нами в ячейке IRQ_Handle с целью дальнейшего использования. Функция VPICD_Virtualize_IRQ фактически устанавливает в системе наш обработчик прерываний, имя которого включено нами в структуру VPICD_IRQ_Descriptor. Начиная с этого момента аппаратные прерывания IRQ5 будут вызывать не обработчик этого уровня по умолчанию, находящийся в VPICD, а наш обработчик. Правда, для этого надо размаскировать уровень 5 в контроллере прерываний, что мы еще не сделали.

При вызове драйвера из приложения 81-01.EXE управление передается API-процедуре драйвера API_Handler. В ней прежде всего извлекаются из структуры клиента переданные в драйвер параметры. Поскольку эти параметры (содержимое регистров клиента) хранятся в стеке уровня 0, т. е. в памяти, их нельзя непосредственно перенести в ячейки данных драйвера. Мы для переноса параметров в некоторых случаях использовали стек, в других – регистры общего назначения.

Выполнив команду общего сброса программируемой платы, следует размаскировать прерывания в физическом (не виртуальном) контроллере прерываний. Эта операция осуществляется вызовом функции виртуального контроллера VPICD_Physically_Unmask с указанием ей в качестве параметра в регистре EAX дескриптора виртуального прерывания. Далее выполняется уже рассмотренная в предыдущей статье процедура инициализации платы (причем значения констант C0, C1 и C2 извлекаются из структуры клиента). После завершения API-процедуры управление возвращается в приложение до поступления аппаратного прерывания.

Аппаратное прерывание виртуализованного нами уровня через дескриптор таблицы прерываний IDT с номером 55h активизирует обработчик прерываний, входящий в состав VPICD, который, выполнив некоторые подготовительные действия (в частности, сформировав на стеке уровня 0 структуру клиента), передает управление непосредственно нашему драйверу, конкретно – процедуре обработки аппаратного прерывания VMyD_Int_13. Системные издержки этого перехода составляют около 40 команд процессора, т. е. время от момента поступления прерывания до выполнения первой команды нашего обработчика составит на компьютере среднего быстродействия 10...15 мкс.

В процедуре VMyD_Int_13 после выполнения содержательной части – в нашем случае чтения и запоминания результата измерений – необходимо послать в контроллер прерываний команду EOI, как это полагается делать в конце любого обработчика аппаратного прерывания. Для виртуализованного прерывания это действие выполняется с помощью функции VPICD_Phys_EOI, единственным параметром которой является дескриптор прерывания, который мы сохранили в ячейке IRQ_Handle. Последней операцией является вызов функции VPICD_Physically_Mask, с помощью которой маскируется уровень 5 в физическом контроллере прерываний.

Следует заметить, что названия функций VPICD могут быть обманчивыми. Функция VPICD_Phys_EOI в действительности не разблокирует контроллер прерываний, а размаскирует наш уровень в регистре маски физического контроллера (чего мы, между прочим, не заказывали!). Что же касается команды EOI, то она была послана в контроллер по ходу выполнения фрагмента VPICD еще до перехода на наш обработчик (упомянутые выше 40 команд). Тем не менее вызов функции VPICD_Phys_EOI в конце обработчика прерываний обязателен. Если ею пренебречь, то система будет вести себя точно так же, как если бы в контроллер не была послана команда EOI: первое прерывание обрабатывается нормально, но все последующие прерывания блокируются. Так происходит потому, что при отсутствии вызова функции VPICD_Phys_EOI нарушается работа функции VPICD_Physically_Unmask, которая выполняется у нас на этапе инициализации. Эта функция, выполнив анализ системных полей и обнаружив, что предыдущее прерывание не завершилось вызовом VPICD_Phys_EOI, обходит те свои строки, в которых в порту 21h устанавливается в 0 бит нашего уровня прерываний. В результате этот уровень остается замаскированным и прерывания не проходят.

Если обработчик прерываний, включенный в драйвер, выполняет только обслуживание аппаратуры, то на этом его программа может быть завершена. Однако мы хотим оповестить о прерывании приложение, вызвав одну из его функций. VMM предусматривает такую возможность (так называемое вложенное выполнение VM), однако для ее реализации следует прежде всего перейти с асинхронного уровня на синхронный.

Проблема заключается в том, что VMM является нереентерабельной программой. Если переход в виртуальный драйвер осуществляется синхронным образом, вызовом из текущего приложения, то, хотя этот переход происходит при участии VMM и, так сказать, "через него", в активизированной процедуре виртуального драйвера допустим вызов всех функций VMM. Если же переход в драйвер произошел асинхронно, в результате аппаратного прерывания, то состояние VMM в этот момент неизвестно и в процедуре драйвера допустим вызов лишь небольшого набора функций, относящихся к категории асинхронных. К ним, в частности, относятся все функции VPICD, а также те функции VMM, с помощью которого программа переводится на синхронный уровень (его иногда называют уровнем отложенных прерываний). В справочнике, входящем в состав DDK, указано, какие функции являются асинхронными, и на эту характеристику функций следует обращать внимание.

Идея перехода на уровень отложенных прерываний заключается в том, что в обработчике аппаратных прерываний с помощью одной из специально предназначенных для этого асинхронных функций VMM устанавливается запрос на вызов callback-функции (функции обратного вызова). Эта функция будет вызвана средствами VMM в тот момент, когда переход не нее на нарушит работоспособность VMM. Вся эта процедура носит название "обработки события".

В понятие события входит не только callback-функция, но и набор условий, при которых она может быть вызвана или которыми должен сопровождаться ее вызов. Так, например, можно указать, что callback-функцию можно вызвать только вне критической секции или что вызов callback-функции должен сопровождаться повышением приоритета ее выполнения. Кроме этого, при установке события можно определить некоторое данное (двойное слово), которое будет передано в callback-функцию при ее вызове. В составе VMM имеется целый ряд функций установки событий, различающихся условиями их обработки, например Call_When_Idle, Call_When_Not_Critical,

Call_Restricted_Event, Schedule_Global_Event, Schedule_Thread_Event и др. Существенно подчеркнуть, что момент фактического вызова callback-функции определить заранее невозможно. Она может быть вызвана немедленно, а может быть вызвана спустя некоторое время, когда будут удовлетворены поставленные условия.

В нашем случае специальные условия отсутствуют и переход на синхронный уровень можно выполнить с помощью простой функции Call_VM_Event, в качестве параметра которой указывается 32-битовое смещение функции обратного вызова, располагаемой в тексте виртуального драйвера. В рассматриваемом примере эта функция названа Reflect_Int.

Команда ret, которой заканчивается обработчик прерываний виртуального драйвера, передает управление VMM, который в удобный для него момент времени вызывает функцию Reflect_Int (реально до вызова может пройти 50...200 мкс). В этой функции вызовом Push_Client_State исходная структура клиента еще раз сохраняется на стеке уровня 0, после чего функцией Begin_Nest_Exec открывается блок вложенного выполнения. Внутри этого блока можно, во-первых, организовать переход на определенную функцию приложения и, во-вторых, создать условия для передачи ей требуемых параметров. Передача параметров осуществляется в соответствии с установленным интерфейсом используемого языка программирования. Поскольку наше приложение написано на языке Си, для его функций действуют правила этого языка – параметры передаются функции через стек, причем расположение параметров в стеке должно соответствовать их перечислению в прототипе и заголовке функции, т. е. в глубине стека должен находиться последний параметр, а на вершине стека – первый (в функциях "типа Паскаля", в частности во всех системных функциях Windows, действует обратный порядок передачи параметров).

Помещение параметров в стек текущей VM осуществляется функцией VMM Simulate_Push, которая может проталкивать в стек как одинарные, так и двойные слова. В нашем случае в стек помещаются два слова – результат измерений и селектор сегмента данных приложения.

Следующая операция – подготовка вызова требуемой функции приложения. Эта операция осуществляется с помощью функции VMM Simulate_Far_Call, которая помещает передаваемые ей в качестве параметров селектор и смещение требуемой функции приложения в поля структуры клиента Client_CS и Client_IP. В результате, когда VMM, передавая управление приложению, снимет со стека структуру клиента и выполнит переход по оставшимся в стеке значениям Client_CS и Client_IP, в регистрах CS:IP окажется адрес интересующей нас функции, которая и начнет немедленно выполняться. Для того чтобы не потерять то место в приложении, на котором произошла его приостановка из-за прихода прерывания, текущее содержимое полей Client_CS и Client_IP сохраняется в созданной перед этим копии структуры клиента.

Наконец, вызовом Resume_Exec управление передается в приложение. Еще раз подчеркнем, что этот вызов функции приложения является вложенным в VMM и возможности вызываемой функции весьма ограничены. Фактически она работает в чуждой для приложения операционной среде. В частности, как уже отмечалось, содержимое сегментных регистров (кроме CS) не соответствует сегментам приложения. Для того чтобы функция isr() могла обратиться к глобальным переменным приложения (адресуемым через регистр DS) мы передаем ей селектор сегмента данных приложения.

Вернемся ненадолго к программе 81-01.CPP. Функция isr(), которую мы вызываем из драйвера, имеет следующий вид:

```
void isr(int segment,int dt){
...
}
```

Поскольку мы в драйвере протолкнули в стек сначала данное Data, а затем селектор DSseg, они расположились в стеке приложения в правильном с точки зрения этой функции порядке и она может обращаться к своим локальным переменным segment и dt, как если бы она была вызвана обычным образом оператором

```
isr(DSseg,Data);
```

После завершения функции isr() управление возвращается в VMM, а из него в драйвер на команду, следующую за вызовом Resume_Exec. Этот переход может потребовать пары сотен команд и нескольких десятков микросекунд.

Отложенная процедура драйвера завершается очевидными вызовами End_Nest _Exec окончания вложенного блока выполнения и Pop_Client_State восстановления структуры клиента.

Описанная методика организации взаимодействия обработчика аппаратных прерываний, включенного в состав драйвера, и самого приложения относительно сложна и, тем не менее, не обеспечивает необходимые функциональные возможности обработчику прерываний приложения, в котором запрещается вызов функций Windows. Для того чтобы по сигналу прерывания вывести на экран результаты измерений, нам пришлось создавать специальный цикл обработки сообщений с постоянным опросом флага request. Установка флага обработчиком прерываний приложения приводила к выполнению функции PostMessage() и посылке в приложение сообщения WM_USER, в ответ на которое мы могли уже выполнять любые программные действия без каких-либо ограничений.

Заметного упрощения программы можно добиться, организовав посылку в приложение сообщения WM_USER непосредственно из обработчика прерываний драйвера, точнее, с уровня отложенных прерываний. В этом случае отпадает необходимость в передаче драйверу каких-либо данных, кроме дескриптора того окна приложения, в которое посылается сообщение WM_USER, в данном случае дескриптора главного окна. Сокращается и текст процедуры отложенных прерываний Reflect_Int. Приложение Windows также упрощается: отпадает необходимость в разделении функций обработки прерываний между функцией Isr(), работающей, по существу, на уровне отложенных прерываний, и функцией OnUser(), выполняемой уже на обычном уровне задачи. Поскольку результат измерений легко передать из драйвера в приложение в качестве параметра сообщения WM_USER, отпадает необходимость в пункте "Чтение" в меню приложения.

В примере 81.2 рассмотрим изменения, вносимые при использовании такого метода в программы примера 81.1.

Пример 81.2. Приложение Windows, обрабатывающее аппаратные прерывания

```
//Операторы препроцессора #define и #include
...
#define HANDLE_WM_USER(hwnd, wParam, lParam, fn) \//Макрос для обработки
        ((fn)(hwnd,wParam), 0L)       // сообщения WM_USER
```

```
//Прототипы функций
...
void OnUser(HWND,WPARAM);  //Сигнатура функции изменилась
//Глобальные переменные
HWND hMainWnd;             //Дескриптор главного окна
...
//Главная функция WinMain
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE,LPSTR,int){
...
  while(GetMessage(&msg,NULL,0,0))//Обычный цикл
    DispatchMessage(&msg);//обработки сообщений
  return 0;
  }//Конец WinMain
//Функция Register регистрации класса окна
...
//Функция Create создания и показа окна
void Create(HINSTANCE hInst){
  hMainWnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
     10,10,200,100,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hMainWnd,SW_SHOWNORMAL);
  }
//Оконная функция WndProc главного окна
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    HANDLE_MSG(hwnd,WM_USER,OnUser);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
//Функция OnCommand обработки сообщений WM_COMMAND от команд меню
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id){
    case MI_START://Инициализация платы
      InitCard();
      break;
    case MI_EXIT://Завершить приложение
      DestroyWindow(hwnd);
    }
  }
//Функция OnUser обработки сообщения WM_USER
void OnUser(HWND,WPARAM wParam){
  char txt [80];
  wsprintf(txt,"Измерения окончены\nНакоплено %d событий",(UINT)wParam);
  MessageBox(hMainWnd,txt,"Контроль",MB_ICONINFORMATION);
  }
//Функция OnDestroy обработки сообщения WM_DESTROY
...
//Функция InitCard инициализации (через драйвер) платы
void InitCard (){
  GetAPIEntry();             //Получение адреса точки входа в драйвер
  _BX=55;                    //Канал 0
  _CX=20000;                 //Канал 1; BX*CX=Длительность интервала
  _DX=1000;                  //Канал 2, внутренний генератор
  _DI=(UINT)hMainWnd;        //Дескриптор главного окна
  VxDEntry();                //Вызов драйвера
  }
//Функция GetAPIEntry получения адреса точки входа в драйвер
void GetAPIEntry()
...
```

В приведенном выше тексте файла .CPP детально показаны только измененные участки программы.

Изменилось определение макроса HANDLE_WM_USER для обработки сообщения WM_USER, которое мы пошлем в приложение из драйвера: функция обработки этого сообщения fn (в приложении она носит название OnUser()) принимает два параметра: hwnd и wParam. Через параметр сообщения wParam в приложение будет передан результат измерений. При необходимости передавать в приложение больший объем данных, можно было расширить состав параметров функции третьим параметром lParam.

Цикл обработки сообщений существенно упростился и принял форму, обычную для простых приложений Windows.

В функции OnCommand() удален фрагмент, связанный с пунктом меню "Чтение" (идентификатор MI_READ), поскольку в этом пункте уже нет необходимости.

В функции OnUser() параметр wParam приводится к типу целого без знака, преобразуется в символьную форму и выводится на экран в окно сообщения с соответствующим текстом.

Как будет видно из программы драйвера (и как, впрочем, должно быть очевидно читателю), при посылке из драйвера сообщения WM_USER необходимо указать системе, какому окну адресовано это сообщение. Однако драйверу, естественно, ничего не известно про окна приложений; дескриптор нашего главного окна следует передать драйверу на этапе инициализации платы. Это выполняется в функции InitCard(), где перед вызовом драйвера в регистры BX, CX и DX засылаются константы настройки таймера, а регистр DI – приведенный к целому типу дескриптор главного окна hMainWnd.

Посмотрим теперь, как изменится программа драйвера (пример 81.2, продолжение).

Пример 81.2 (продолжение). Программа драйвера для обслуживания аппаратных прерываний

```
...
WM_USER=SPM_UM_AlwaysSchedule+400h;Код сообщения WM_USER
include shell.inc          ;Дополнительный включаемый файл
...
;======================
VxD_DATA_SEG
Data dw 0,0                ;32-битовая ячейка с данным для передачи в приложение
hwnd dd 0                  ;32-битовая ячейка для получения дескриптора окна
IRQ_Handle dd  0           ;Дескриптор виртуального прерывания
VMyD_Int13_Desc label dword;32-битовый адрес следующей далее структуры
VPICD_IRQ_Descriptor <5,,OFFSET32 VMyD_Int_13>;Структура с данными о прерывании
VxD_DATA_ENDS
;======================
VxD_CODE_SEG
BeginProc VMyD_Control
...
EndProc VMyD_Control
;----------------------
BeginProc VMyD_Device_Init
...
EndProc VMyD_Device_Init
;------------------------
;API-процедура, вызываемая из приложения
;При вызове: BX=C0, CX=C1, DX=C2, DI=дескриптор главного окна
BeginProc API_Handler
;Получим параметры из приложения
        movzx  EAX,[EBP.Client_DI]
```

```
        mov    hwnd,EAX
;Общий сброс
...
;Размаскируем уровень 5 в физическом контроллере прерываний
...
;Засылаем управляющие слова по каналам
...
;Засылаем константы в каналы
...
;Установим флаг S2 разрешения счета
...
        ret
EndProc API_Handler
;-------------------------
;Процедура обработки аппаратного прерывания IRQ5 (вектор 13)
BeginProc VMyD_Int_13, High_Freq
;Получим результат измерений из выходного регистра счетчика
        ...
        mov    Data,AX      ;Результат в младшей половине Data
;Выполним завершающие действия в PIC и вызовем функцию приложения
        mov    EAX,IRQ_Handle
        VxDCall VPICD_Phys_EOI;EOI в физический контроллер прерываний
        VxDCall VPICD_Physically_Mask;Маскируем наш уровень
;Перейдем на синхронный уровень. Это все иначе
        push   0            ;Тайм-аут
        push   CAAFL_RING0 ;Событие кольца 0
        push   0            ;Данные для передачи в процедуру отложенных прерываний
        push   OFFSET32 Reflect_Int;Вызываемая процедура отложенных прерываний
        VxDCall _SHELL_CallAtAppyTime;Вызвать во "время приложения"
        add    ESP,4*4      ;Компенсация стека
        clc
        ret
EndProc VMyD_Int_13
;-------------------------
;Процедура уровня отложенных прерываний. Это тоже иначе
BeginProc Reflect_Int
        push   0            ;Данные для функции обратного вызова
        push   0            ;Адрес функции обратного вызова
        push   0            ;lParam
        push   Data         ;wParam
        push   WM_USER      ;Код сообщения
        push   hwnd         ;Окно-адресат
        VxDCall _SHELL_PostMessage;Поставить сообщение в очередь
        add    ESP,4*6      ;Компенсация стека
        clc
        ret
EndProc Reflect_Int
VxD_CODE_ENDS
end VMyD_Real_Init
```

В начале текста драйвера необходимо подключить еще один заголовочный файл SHELL.INC и определить значение константы WM_USER. В Windows эта константа имеет длину 16 бит и равна 400h, однако функции _SHELL_PostMessage необходимо передать 32-битовое слово, причем сам код сообщения WM_USER должен находиться в младшей половине этого слова, а в старшую половину следует поместить информацию о диспетчеризации. В нашем случае эта информация выглядит в виде константы SPM_UM_AlwaysSchedule.

В сегменте данных удалены ячейки для адреса функции обратного вызова isr и селектора DS. Ячейка для результата измерений объявлена как два слова, так как все параметры функции Shell_PostMessage имеют размер 32 бита. Добавлена ячейка hwnd для получения в нее из приложения дескриптора главного окна. Сам дескриптор имеет размер 16 бит, однако передавать его той же функции Shell_PostMessage надо в виде длинного слова.

В начале API-процедуры из структуры клиента (конкретно – из регистра DI) извлекается дескриптор окна и после расширения до длинного слова помещается в ячейку hwnd.

Остальные изменения касаются лишь способа перехода на уровень отложенных прерываний и состава процедуры ReflectInt, работающей на этом уровне.

Для перехода на синхронный уровень в данном случае используется системный вызов _SHELL_CallAtAppyTime, осуществляющий передачу управления указанной в вызове процедуре ReflectInt во "время приложения", т. е. когда управление будет возвращено из VMM в приложение. В этой процедуре уже можно будет поставить сообщение WM_USER в очередь сообщений главного окна нашего приложения.

В процедуре уровня отложенных прерываний ReflectInt после проталкивания в стек необходимых параметров вызывается системная функция _SHELL_PostMessage, которая и посылает в приложение сообщение WM_USER. Легко видеть, что программист должен в этом случае полностью сформировать весь состав структуры сообщения msg – дескриптор окна-адресата, код сообщения, а также оба параметра, входящие во все сообщения Windows, – wParam и lParam. Параметром wParam мы в данном примере пользуемся, чтобы передать в приложение результат измерения из ячейки Data. При необходимости можно было использовать и lParam.

Функция обратного вызова, адрес которой можно было указать в числе параметров, вызывается Windows после успешной постановки в очередь посылаемого сообщения. Мы эту функцию не используем.

Для задач управления аппаратурой, работающей в режиме прерываний, важной характеристикой является время отклика на прерывание, т. е. временная задержка от момента поступления прерывания до выполнения первой команды обработчика. Как мы видели, при использовании виртуального драйвера системные издержки перехода на прикладной обработчик, включенный в состав драйвера, составляют около 40 команд, на выполнение которых на машине средней производительности может понадобиться 10...15 мкс. При использовании системы MS-DOS этих издержек не было бы вовсе, так как в реальном режиме переход на обработчик прерываний осуществляется процессором аппаратно и, следовательно, практически мгновенно. Если же реализовать обработку прерываний без помощи виртуального драйвера, как это было сделано в примере 80.1, то переход на прикладной обработчик прерываний потребовал бы 200...300 команд, а время задержки увеличилось бы (на таком же компьютере) до 120...180 мкс, т. е. более чем на порядок. С другой стороны, использование уровня отложенных прерываний может существенно и, что немаловажно, непредсказуемым образом увеличить время отклика.

Статья 82. **Диагностический вывод информации из драйвера**

В ряде случаев, главным образом в процессе отладки нового драйвера или при исследовании его работы, оказывается удобным не передавать данные из виртуального драйвера в вызывающее приложение, а непосредственно вывести эти данные из драйвера на экран. Приведем пример драйвера, который при вызове его из приложения защищенного режима выводит на экран характерные данные виртуальной машины: номер виртуальной машины, ее дескриптор, а также адрес структуры клиента (пример 82.1).

Пример 82.1. Диагностический вывод информации из драйвера. Файл 82-01.ASM исходного текста виртуального драйвера

```
.386p
.XLIST
include vmm.inc
include shell.inc         ;Поддержка средств вывода сообщений
.LIST
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h, \
     Undefined_Init_Order,,API_Handler
;========================
VxD_REAL_INIT_SEG
BeginProc VMyD_Real_Init
        mov   AH,09h
        mov   DX,offset msg
        int   21h
        mov   ax, Device_Load_Ok
        xor   bx, bx
        xor   si, si
        xor   edx, edx
ret
msg db 'Виртуальный драйвер VMyD загружен$'
EndProc   VMyD_Real_Init
VxD_REAL_INIT_ENDS
;======================
VxD_DATA_SEG
Caption db 'Message #1',0
Mesg    db 'VM ID='
VMID    db '****  ',13,10
        db 'VM Handler='
VMHandle db '********  ',13,10
        db 'Client Struct='
ClSt    db '********',0
VxD_DATA_ENDS
;======================
VxD_CODE_SEG
BeginProc VMyD_Control
        clc
        ret
EndProc VMyD_Control
;-------------------------
;API-процедура защищенного режима
BeginProc API_Handler
;Получим и выведем номер VM
        mov   EAX,[EBX].CB_VMID;Получили номер VM (фактически в AX)
        mov   ESI,offset32 VMID;Адрес строки
        call  Word_ascii  ;Преобразуем в символьную форму
```

```
;Получим и выведем дескриптор VM
        mov     EAX,EBX      ;Получили дескриптор VM
        mov     ESI,offset32 VMH+4;Адрес строки
        call    Word_ascii   ;Преобразуем в символьную форму
        shr     EAX,16       ;Сдвинем старшую половину данного в AX
        mov     ESI,offset32 VMH;Адрес строки
        call    Word_ascii
;Получим и выведем адрес структуры клиента
        mov     EAX,[EBX].CB_Client_Pointer
        mov     ESI,offset32 ClSt+4;Адрес строки
        call    Word_ascii   ;Преобразуем в символьную форму
        shr     EAX,16       ;Сдвинем старшую половину данного в AX
        mov     ESI,offset32 ClSt;Адрес строки
        call    Word_ascii   ;Преобразуем в символьную форму
        call    Info         ;Выведем всю строку на экран
        ret
EndProc API_Handler
;---------------------------
;Процедура вызова функции вывода сообщения
BeginProc Info
        mov     EAX,MB_OK    ;Константа формата сообщения
        mov     ECX,offset32 Mesg;Адрес выводимой строки
        mov     EDI,offset32 Caption;Заголовок сообщения драйвера
        VxdCall SHELL_Sysmodal_Message
        ret
EndProc Info
;---------------------------
BeginProc Word_ascii
        push    AX           ;Сохраним исходное данное
        and     AX,0F000h    ;Выделим старшую четверку битов
        shr     AX,12        ;Сдвиг вправо до начала регистра
        call    Bin_ascii    ;Преобразуем в символ
        mov     byte ptr [ESI],AL;Отправим его в строку
        inc     ESI          ;Сдвиг по строке символов
        pop     AX           ;Вернем в AX исходное данное
        push    AX           ;Сохраним его
        and     AX,0F00h     ;Выделим следующую четверку битов
        shr     AX,8         ;Сдвиг вправо до начала регистра
        call    Bin_ascii    ;Преобразуем в символ
        mov     byte ptr [ESI],AL;Отправим его в строку
        inc     ESI          ;Сдвиг по строке символов
        pop     AX           ;Вернем в AX исходное данное
        push    AX           ;Сохраним его
        and     AX,0F0h      ;Выделим следующую четверку битов
        shr     AX,4         ;Сдвиг вправо до начала регистра
        call    Bin_ascii    ;Преобразуем в символ
        mov     byte ptr [ESI],AL; Отправим его в строку
        inc     ESI          ;Сдвиг по строке символов
        pop     AX           ;Вернем в AX исходное данное
        and     AX,0Fh       ;Выделим младшую четверку битов
        call    Bin_ascii    ;Преобразуем в символ
        mov     byte ptr [ESI],AL;Отправим его в строку
        ret
EndProc Word_ascii
;---------------------------
BeginProc Bin_ascii
        cmp     AL,9         ;Цифра > 9?
        ja      letter       ;Да, на преобразование в символы A...F
        add     AL,30h       ;Нет, преобразуем в символ 0...9
        jmp     ok           ;И на выход
letter: add     AL,37h       ;Преобразуем в символы A...F
```

```
ok:     ret
EndProc Bin_ascii
VxD_CODE_ENDS
end VMyD_Real_Init
```

Вывод сообщения из драйвера на экран осуществляется функцией SHELL_Sysmodal_Message, вызов которой в рассматриваемом примере оформлен в виде процедуры Info. Функция SHELL_Sysmodal_Message вызывается с помощью макроса VxDCall, определенного в файле VMM.INC. В качестве параметров функции указывается адрес символьной строки, выводимой на экран (в регистре ECX), а также адрес строки с заголовком сообщения (в EDI). Функция требует наличия в регистре EBX дескриптора виртуальной машины, однако при вызове драйвера из приложения этот дескриптор заносится в EBX автоматически и о нем можно не заботиться. В регистре EAX с помощью различных символических констант можно указать формат сообщения. Все эти константы определены в файле SHELL.INC, который необходимо подключить к исходному тексту драйвера директивой include. Константа MB_OK, использованная в примере, предполагает продолжение работы драйвера (в частности, возврат в вызвавшее драйвер приложение) после нажатия пользователем клавиши Enter; при указании других констант можно организовать диалог драйвера с пользователем. Например, при использовании в качестве параметра константы MB_YESNO в сообщение драйвера включается указание на альтернативное продолжение по нажатию клавиш Y (да) или N (нет). Реакция драйвера на ту или иную команду предусматривается в его тексте таким образом:

```
BeginProc Info
        mov    EAX,MB_YESNO   ;Константа формата сообщения
        mov    ECX,offset32 Mesg;Адрес выводимой строки
        mov    EDI,offset32 Caption;Заголовок сообщения драйвера
        VxdCall SHELL_Sysmodal_Message
        cmp    EAX,IDYES
        je     yyy            ;Переход при нажатии клавиши Y
        jmp    nnn            ;Переход при нажатии клавиши N
        ret
EndProc Info
```

Строка Mesg, составляющая содержание выводимого на экран сообщения, для удобства заполнения ее конкретными данными разбита с помощью последовательности директив db на несколько символьных полей. Тем полям, в которые предполагается поместить данные, присвоены имена (VMID, VMHandle и ClSt). Начальное заполнение этих полей знаком звездочки * может помочь в отладке программы драйвера.

Для преобразования данных, представленных в виде двоичных чисел, в символьные строки в состав драйвера включены две подпрограммы. Процедура Bin_ascii преобразует четверку битов в символ 16-ричной системы счисления; процедура Word_ascii преобразует 2-байтовое число, находящееся в регистре AX, в символьную форму (с использованием для преобразования каждой четверки битов подпрограммы Bin_ascii) и заносит его память по адресу, находящемуся в регистре ESI. Обе эти процедуры оформлены с помощью макросов BeginProc и EndProc. Алгоритм преобразования чисел в символы рассматривался в статье 13; в тексты процедур примера 13.1 внесены незначительные изменения, чтобы приспособить их к работе в 32-разрядной среде виртуального драйвера.

Дескриптор виртуальной машины, который мы хотим вывести на экран, при вызове драйвера помещается системой в регистр EBX, а остальные интересующие нас дан-

ные входят в состав управляющего блока, адресом которого как раз и является дескриптор VM. Все эти данные имеют определенные символические обозначения (см. статью 74). Заполнение строки Mesg данными в символьной форме начинается в программе с идентификатора (номера) VM. Этот номер формально занимает в блоке управления двойное слово, однако фактически находится в младшей половине этого слова. После переноса номера VM из структуры, адресуемой через EBX, в регистр EAX в ESI загружается адрес поля VMID в строке Mesg и вызывается процедура Word_ascii, выполняющая преобразование номера VM в символьную форму и занесение результата преобразования в 4 байта поля VMID.

Далее в регистр EAX загружается дескриптор VM (из регистра EBX), в регистр ESI – адрес VMH+4 правой четверки байтов поля VMH и вызовом процедуры Word_ascii (которая работает с регистром AX, т. е. с младшей половиной дескриптора) эта младшая половина помещается на свое место в символьной строке. Старшая половина дескриптора командой shr сдвигается в регистр AX, преобразуется в символы и заполняет левую четверку байтов поля VMH, в результате чего мы наблюдаем на экране 32-битовое число в привычном для нас виде (старшие цифры слева, младшие – справа).

Аналогично выполняется преобразование адреса структуры клиента. Последней командой процедуры API_Handler вызывается подпрограмма Info, которая выводит сформированную строку Mesg на экран. Сообщение драйвера выглядит приблизительно так, как показано на рис. 82.1.

```
                    Message #1
VM ID=0001
VM Handle=C39200E8
Client Struct=C0F22F70
```

Рис. 82.1. Пример сообщения драйвера

Для того чтобы процедура API_Handler начала выполняться, ее следует вызывать из приложения Windows. Возможный текст такого приложения приведен ниже (пример 82.1, продолжение). Оно имеет, можно сказать, вырожденный характер, так как в нем не формируется никаких окон или других изобразительных элементов Windows, а только определяется адрес драйвера и выполняется его вызов и затем для контроля хода программы на экран выводится окно сообщения с текстом "Драйвер вызван". В сущности, последнее действие излишне.

Пример 82.1 (продолжение). Диагностический вывод информации из драйвера. Файл 82-01.CPP приложения Windows

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
void GetAPIEntry();
FARPROC VxDEntry;
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  GetAPIEntry();
  VxDEntry();
  MessageBox(NULL,"Драйвер вызван","Info",MB_ICONINFORMATION);
  return 0;
  }
  void GetAPIEntry(){
  asm{
    mov  AX,0x1684
    mov  BX,0x8000
```

```
int  0x2F
mov  word ptr VxDEntry,DI
mov  word ptr VxDEntry+2,ES
 }
}
```

Статья 83. **Взаимодействие драйвера с 32-разрядным приложением Windows**

В 16-разрядных приложениях Windows (как и в приложениях DOS) для получения адреса API-процедуры прикладного виртуального драйвера используется прерывание 2Fh (функция 1684h). Попытка воспользоваться той же процедурой в 32-разрядном приложении Windows приведет к аварийному завершению приложения: в 32-разрядных приложениях недопустим вызов широко используемых в DOS и в 16-разрядных приложениях Windows программных прерываний общего назначения 2Fh, 21h и ряда других. Запрет этот связан с тем, что обработчики этих прерываний, входящие в состав Windows, являются 16-разрядными программами, которые по ходу своего выполнения используют контекст прерванного приложения, в частности работают с его стеком. При этом обращение к стеку часто осуществляется через регистр BP, в который сначала переносится содержимое регистра SP. Однако содержимое SP в данной ситуации лишено смысла, так как отражает лишь младшую половину истинного смещения в стеке, которое в 32-разрядном приложении находится в расширенном регистре ESP. Первое же обращение к стеку через регистр BP адресует программу в случайное место стека, и дальнейший ход программы неминуемо нарушается.

Другое, более тонкое обстоятельство, не позволяющее использовать прерывания 2Fh и 21h в 32-разрядных приложениях, связано с содержимым дескрипторов этих прерываний, входящих в состав таблицы дескрипторов IDT прерываний защищенного режима. Дескрипторы прерываний 2Fh и 21h описаны как шлюзы 286-процессора (в отличие от дескрипторов многих других программных и всех аппаратных прерываний, являющихся шлюзами 386-процессора). Тип дескриптора прерывания влияет на процедуру выполнения команды int. Переход на обработчик прерывания через шлюз 386-го процессора сопровождается сохранением в стеке трех двойных слов (EFLAGS, CS и EIP) и может использоваться как в 16-разрядных, так и в 32-разрядных приложениях. Для шлюзов 286-процессора команда int действует так же, как и в реальном режиме, сохраняя в стеке 16-разрядные регистры: флаги, CS И IP. Поскольку при этом старшая часть расширенного указателя команд EIP не сохраняется, теряется возможность возврата из обработчика прерывания в вызвавшее это прерывание 32-разрядное приложение.

Для связи 32-разрядных приложений Windows с виртуальными драйверами предусмотрен так называемый IOCTL-интерфейс (от Device In-Out Control). Приложение, вызывая специально предусмотренную функцию Windows DeviceIoControl(), заставляет Windows послать в заданный драйвер системное сообщение W32_DeviceIOControl, которое включает в себя условный код требуемого действия, а также адреса входного и выходного буферов для передачи данных. Драйвер, получив это сообщение, определяет, какое именно действие (если их несколько) ему следует выполнить. При этом драйвер имеет возможность через входной буфер, оговоренный в IOCTL-интерфейсе, получить из приложения необходимую информацию, а через выходной передать в приложение результаты своей работы. Структура клиента, с помощью которой мы пе-

редавали данные из приложения в драйвер и обратно, в 32-разрядных приложениях не используется.

Реализация IOCTL-интерфейса в вызывающем приложении Windows состоит из трех этапов: открытия драйвера с получением его дескриптора; посылки, с помощью вызова функции DeviceIoControl(), сообщения W32_DeviceIOControl с указанием кода требуемого действия и закрытия драйвера с освобождением выделенного для связи с ним дескриптора.

Открытие драйвера осуществляется 32-разрядной файловой функцией CreateFile(), которая для этого случая имеет вид:

```
HANDLE
    xVxd=CreateFile("\\\\.\\VMYD",0,0,NULL,0,FILE_FLAG_DELETE_ON_CLOSE,NULL);
```

где VMYD – имя виртуального драйвера (без расширения); hVxd – переменная типа HANDLE, в которую функция возвращает дескриптор открытого драйвера. Если по тем или иным причинам драйвер открыть не удалось, функция CreateFile() вместо дескриптора драйвера (небольшого числа, например 2) возвращает в качестве кода ошибки значение INVALID_HANDLE_VALUE (равное 0xFFFFFFFF).

Вызов функции DeviceIoControl может выглядеть следующим образом:

```
DeviceIoControl (hVxd,DIOC_FUNC,&inBuf,inSize,&outBuf,outSize,&cb,0);
```

где hVxd – дескриптор виртуального драйвера; DIOC_FUNC – код требуемой функции (его значение будет анализировать API-процедура драйвера); inBuf – имя программного буфера, в котором приложение подготавливает данные, передаваемые в драйвер. В документации по IOCTL-интерфейсу буферам обмена данными даются названия с точки зрения драйвера. Так, inBuf – это входной для драйвера (и, естественно, выходной для приложения) буфер, а outBuf – буфер, выходной для драйвера и входной для приложения. (В вызове DeviceIoControl() указывается адрес буфера, на что указывает значок амперсанта – &inBuf.); inSize – размер буфера inBuf, т. е. число передаваемых в драйвер байтов; outBuf – имя программного буфера, в который драйвер поместит передаваемые им в приложение данные; outSize – размер буфера outBuf, т. е. число байтов, передаваемых в приложение; cb – имя переменной длиной 32 бита, в которую после выполнения функции будет возвращена фактическая длина передаваемых в приложение данных.

Закрытие драйвера осуществляется стандартной файловой функцией CloseHandle (hVxd) с указанием закрываемого дескриптора.

Рассмотрим теперь особенности виртуального драйвера, использующего IOCTL-интерфейс. Как уже отмечалось ранее (см. статью 75), в блоке описания драйвера указываются адреса трех процедур: процедуры для обработки сообщений Windows, API-процедуры, вызываемой из приложений DOS, и API-процедуры, вызываемой из 16-разрядных приложений защищенного режима. Поскольку IOCTL-интерфейс 32-разрядных приложений реализуется с помощью посылки драйверу сообщения Windows (конкретно – W32_DeviceIOControl), то прием этого сообщения необходимо выполнить в процедуре VMyD_Control. API-процедуры для связи с 32-разрядными приложениями не используются; если, однако, драйвер носит универсальный характер и предназначен для взаимодействия со всеми тремя видами приложений (DOS, 16-разрядные Windows и 32-разрядные Windows), то в нем должны быть и соответствующие API-процедуры. В наших примерах эти процедуры будут отсутствовать.

Для диспетчеризации поступившего в драйвер сообщения удобно воспользоваться макросом ControlDispatch:

```
BeginProc VMyD_Control
Control_Dispatch W32_DeviceIOControl,IOControl
        clc
        ret
EndProc VMyDControl
```

Здесь IOControl – произвольное имя прикладной процедуры, предназначенной для обработки запросов драйверу. В приведенном фрагменте поступление сообщения W32_DeviceIOControl приведет к передаче управления процедуре IOControl; все остальные сообщения Windows не обрабатываются. В течение своей жизни Windows посылает каждому драйверу, установленному в системе, целый ряд сообщений: Device_Init – об инициализации данного драйвера, Sys_VM_Init – об инициализации системной виртуальной машины, Sys_VM_Terminate – об ее завершении, Begin_PM_App – о запуске приложения защищенного режима и др. Если предполагается обрабатывать какие-либо из этих сообщений Windows, то для каждого сообщения в процедуру VMyD_Control следует включить свой макрос Control_Dispatch с символическим кодом сообщения и именем прикладной процедуры для его обработки.

Как уже отмечалось, сообщение W32_DeviceIOControl содержит в себе номер требуемого действия. В процессе открытия драйвера вызовом CreateFile() Windows посылает в открываемый драйвер сообщение W32_DeviceIOControl с кодом действия DIOC_GETVERSION (значение этого кода равно нулю), чтобы определить, поддерживает ли драйвер IOCTL-интерфейс. Драйвер в ответ на получение кода действия DIOC_GETVERSION обязан вернуть нулевое значение в регистре EAX, иначе система будет считать, что драйвер не поддерживает IOCTL-интерфейс, и драйвер открыт не будет, а функция CreateFile() вернет код ошибки.

Сообщение W32_DeviceIOControl сопровождается передачей в драйвер через регистр ESI адреса структуры DIOCParams, формируемой системой. Эта структура описана в файле vwin32.inc (который необходимо подключить к исходному тексту драйвера директивой include); она содержит 12 членов, из которых нас будут интересовать следующие:

- VMHandle – дескриптор виртуальной машины;
- dwIOControlCode – условный код требуемой функции;
- lpvInBuffer – адрес буфера приложения, в котором находятся данные, передаваемые в драйвер (входные для драйвера);
- cbInBuffer – размер буфера со входными данными;
- lpvOutBuffer – адрес буфера приложения, в который драйвер может переслать свои выходные данные;
- cbOutBuffer – размер буфера с выходными данными;
- lpoOverlapped – адрес структуры типа OVERLAPPED, используемой в асинхронных операциях.

Драйвер, получив через регистр ESI адрес структуры DIOCParams, может обращаться к ее членам с помощью перечисленных выше обозначений. Мы видим, что в процессе обработки сообщения W32_DeviceIOControl драйверу доступны оба буфера приложения, и с входными и с выходными данными, и обмен данными с приложением может осуществляться путем прямых пересылок. Разумеется, и драйвер и приложение

должны при работе с пересылаемыми данными использовать одинаковые форматы данных.

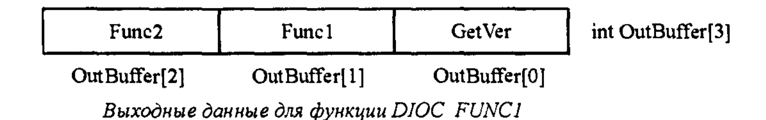
Рассмотрим программный комплекс (пример 83.1), в котором связь драйвера с 32-разрядным приложением Windows организуется посредством IOCTL-интерфейса. Предусмотрим в драйвере обработку трех кодов действия: системного кода DIOC_GETVERSION, поступающего в драйвер в процессе его открытия, и прикладных кодов DIOC_FUNC1 со значением 1 и DIOC_FUNC2 со значением 2 (значения выбраны произвольно).

Получив код действия DIOC_GETVERSION, драйвер сообщает системе о поддержке им IOCTL-интерфейса возвратом EAX=0.

В ответ на получение кода действия DIOC_FUNC1 драйвер переходит на процедуру с именем Func1, которая пересылает в приложение линейные адреса всех трех процедур драйвера. Это чрезвычайно полезная информация, которая позволяет, запустив приложение в отладчике SoftICE, установить точку останова на требуемой процедуре драйвера и исследовать ее выполнение. Можно рекомендовать всегда включать такую процедуру в состав разрабатываемого драйвера. Знание линейных адресов API-процедур драйвера существенно помогает в его отладке.

В ответ на код DIOC_FUNC2 драйвер получает от системы и пересылает в приложение содержимое дескрипторов сегментов команд и данных приложения: их линейные адреса, пределы и атрибуты. В отличие от примера 76.1, где для получения дескриптора мы с помощью относительно трудоемкой процедуры определяли его линейный адрес, в данном примере для этого использована предусмотренная в VMM функция _GetDescriptor.

При разработке виртуального драйвера необходимо заранее оговорить формат данных, поступающих в него из приложения и возвращаемых им в приложение. Поскольку все входные данные должны размещаться в едином буфере ввода, а все выходные данные – в едином буфере вывода, их следует упаковать таким образом, чтобы каждый пакет данных имел определенное имя. В рассматриваемом примере эта проблема была решена так, как показано на рис. 83.1. Выходной буфер для функции DIOC_FUNC1 должен вмещать три линейных адреса; он объявлен как массив из трех длинных слов типа int (напомним, что в 32-разрядных приложениях тип int обозначает 32-битовое данное). Входной буфер для функции DIOC_FUNC2 должен содержать два селектора размером в слово каждый; под них выделено одно длинное слово типа int. Наконец, через выходной буфер для функции DIOC_FUNC2 будут переданы два 8-байтовых дескриптора; каждый дескриптор описывается в приложении с помощью структуры DESCR (как и в примере 76.1), а для размещения двух дескрипторов предусмотрен двухэлементный массив таких структур. Приведенные на рис. 83.1 обозначения OutBuffer[3], InBuffer и sr[2] будут использованы в приложении Windows, написанном на языке Си; с точки зрения драйвера эти буферы представляют собой просто безымянные последовательности байтов, обращение к которым становится возможным по адресам, передаваемым драйверу в структуре DIOCParams.



Выходные данные для функции DIOC_FUNC1

Старшее слово	Младшее слово	
DS	CS	int InBuffer

Входные данные для функции DIOC_2

Старшие 8 байт	Младшие 8 байт	
Дескриптор сегмента данных	Дескриптор сегмента команд	DESCR sr[2]
sr[1]	sr[0]	

Выходные данные для функции DIOC_FUNC2

Рис. 83.1. Форматы пакетов входных и выходных (с точки зрения драйвера) данных в приложении IOCTL32

Рассмотрим сначала текст драйвера (пример 83.1).

Пример 83.1. IOCTL-интерфейс. Файл 83-01.ASM виртуального драйвера

```
.386p
.XLIST
include vmm.inc
include vwin32.inc          ;Поддержка IOCTL-интерфейса
.LIST
DIOC_FUNC1=1                ;Символические обозначения
DIOC_FUNC2=2                ;номеров кодов действия
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h,Undefined_Init_Order
;======================
VxD_REAL_INIT_SEG
BeginProc VMyD_Real_Init
        mov    AH,09h
        mov    DX,offset msg
        int    21h
        mov    ax, Device_Load_Ok
        xor    bx, bx
        xor    si, si
        xor    edx, edx
        ret
msg db 'Виртуальный драйвер VMyD загружен$'
EndProc   VMyD_Real_Init
VxD_REAL_INIT_ENDS
;======================
VxD_DATA_SEG
CSReg   dd 0                ;Ячейки для временного хранения
DSReg   dd 0                ;селекторов, полученных из приложения
VxD_DATA_ENDS
;======================
VxD_CODE_SEG
BeginProc VMyD_Control
Control_Dispatch W32_DeviceIOControl,IOControl
        clc
        ret
EndProc VMyD_Control
```

```
BeginProc IOControl
;Анализ получаемых из приложения (в структуре DIOCParams) кодов действия
        cmp    dword ptr [ESI.dwIOControlCode],DIOC_GETVERSION
        je     GetVer
        cmp    dword ptr [ESI.dwIOControlCode],DIOC_FUNC1
        je     Func1
        cmp    dword ptr [ESI.dwIOControlCode],DIOC_FUNC2
        je     Func2
        clc
        ret
GetVer: xor    EAX,EAX
        clc
        ret
Func1:  mov    EDI,dword ptr [ESI.lpvOutBuffer]
        mov    [EDI],offset32 GetVer
        mov    [EDI+4],offset32 Func1
        mov    [EDI+8],offset32 Func2
        clc
        ret
Func2:  mov    EDI,dword ptr [ESI.lpvInBuffer]
;Примем параметры из приложения (CS, DS)
        movzx  EAX,word ptr [EDI]
        mov    CSReg,EAX
        movzx  EAX,word ptr [EDI+2]
        mov    DSReg,EAX
;Получим дескриптор CS
        VMMCall Get_Cur_VM_Handle
        push   0             ;Флаги
        push   EBX           ;VM дескриптор
        push   CSReg         ;Селектор CS
        VMMCall _GetDescriptor;Получим дескриптор
        add    ESP,3*4       ;Подправим стек
;Передадим весь дескриптор в приложение
        mov    EDI,dword ptr [ESI.lpvOutBuffer]
        mov    [EDI],EAX
        mov    [EDI+4],EDX
;Получим дескриптор DS
        push   0             ;Флаги
        push   EBX           ;VM дескриптор
        push   DSReg         ;Селектор DS
        VMMCall _GetDescriptor;Получим дескриптор
        add    ESP,3*4       ;Подправим стек
;Передадим весь дескриптор в приложение
        mov    EDI,dword ptr [ESI.lpvOutBuffer]
        mov    [EDI+8],EAX
        mov    [EDI+12],EDX
        clc
        ret
EndProc IOControl
VxD_CODE_ENDS
end VMyD_Real_Init
```

В начале текста драйвера определены числовые значения символических обозначений кодов действия DIOC_FUNC1 и DIOC_FUNC2.

В сегменте данных драйвера предусмотрены две ячейки, CSReg и DSReg, для временного хранения селекторов CS и DS, полученных из приложения. Размер этих ячеек (двойные слова) обусловлен тем, что, хотя селекторы имеют длину 16 бит, функция VMM _GetDescriptor требует передачи ей значения селектора в двойном слове (в котором старшие 16 бит игнорируются).

В управляющей процедуре VMyD_Control с помощью макроса Control_Dispatch указано имя IOControl процедуры драйвера, которая предназначена для обработки сообщения W32_DeviceIOControl. В самой процедуре IOControl выполняется анализ поступившего в драйвер кода действия и переход, в зависимости от значения этого кода, на соответствующую процедуру драйвера.

Процедура GetVer возвращает 0 в регистре EAX, сообщая тем самым Windows, что данный драйвер поддерживает IOCTL-интерфейс.

В процедуре Func1 сначала в регистр EDI помещается адрес выходного буфера, а затем с помощью косвенной адресации через EDI со смещением в этот выходной буфер, т. е., в сущности, непосредственно в вызывающее приложение, передаются три смещения процедур – GetVer, Func1 и Func2. Драйвер работает в плоском 4-гигабайтовом адресном пространстве, и указанные смещения представляют собой линейные адреса этих процедур.

В процедуре Func2 таким же способом (через регистр EDI со смещением) из входного буфера последовательно извлекаются два слова, представляющие собой селекторы сегментов приложения CS и DS. Поскольку, как уже отмечалось, для функции VMM _GetDescriptor входные параметры должны быть двойными словами, получаемые параметры тут же командой movzx преобразуются в двойные слова и сохраняются в предусмотренных для них ячейках сегмента данных драйвера.

В предыдущих примерах мы использовали то обстоятельство, что при переходе в драйвер средствами 16-разрядного интерфейса (функция 1684h прерывания 2Fh) VMM загружает в регистр EBX дескриптор виртуальной машины. При вызове драйвера из 32-разрядного приложения с помощью IOCTL-интерфейса этого не происходит и для получения дескриптора VM (который необходим для передачи его в качестве параметра в функцию _GetDescriptor) следует вызвать функцию VMM Get_Cur_VM_Handle, которая возвращает дескриптор в регистре EBX. Другой способ получения дескриптора VM – извлечение его из поля VMHandle структуры DIOCParams.

Для получения дескрипторов сегментов в данном примере использована функция VMM _GetDescriptor. Согласно справочнику DDK, формат ее вызова

```
VMMCall _GetDescriptor, <Selector, VM, flags>
```

предполагает указание параметров функции в виде конкретных чисел. У нас значения селекторов находятся в ячейках памяти CSReg и DSReg, а дескриптор VM – в регистре EBX. Поэтому нам придется воспользоваться расширенной формой вызова этой функции, в которой ее параметры (от правого к левому) проталкиваются в стек явным образом. После возврата из функции (она возвращает младшую половину дескриптора в регистре EAX, а старшую – в EDX) так же "вручную" следует восстановить стек, прибавив к содержимому ESP число байтов, равное суммарному размеру параметров функции.

Полученные дескрипторы передаются в приложение точно так же, как в процедуре Func1 передавались линейные адреса функций. Следует подчеркнуть, что, хотя предложения извлечения из структуры DIOCParams адреса выходного буфера в обеих функциях выглядят одинаково:

```
mov    EDI,dword ptr [ESI.lpvOutBuffer]
```

в действительности в каждом случае через структуру DIOCParams передаются адреса различных буферов приложения, именно тех, которые указаны в соответствующем вызове драйвера DeviceIoControl.

Перейдем теперь к рассмотрению приложения Windows, предназначенного для работы в паре с драйвером 83-01. Тридцатидвухразрядные приложения Windows можно подготовить в среде разработки Borland C++ 4.5, однако удобнее воспользоваться версией 5.0 (или более поздней). С точки зрения пользователя эта версия имеет некоторые отличия от рассмотренной ранее версии 4.5. Так, создание нового проекта выполняется с помощью команды File➤New➤Project. В окне New Target (рис. 83.2) следует установить тип мишени – Application [exe], в окне Platform – Win32. В этом случае будет создано 32-разрядное приложение Windows.

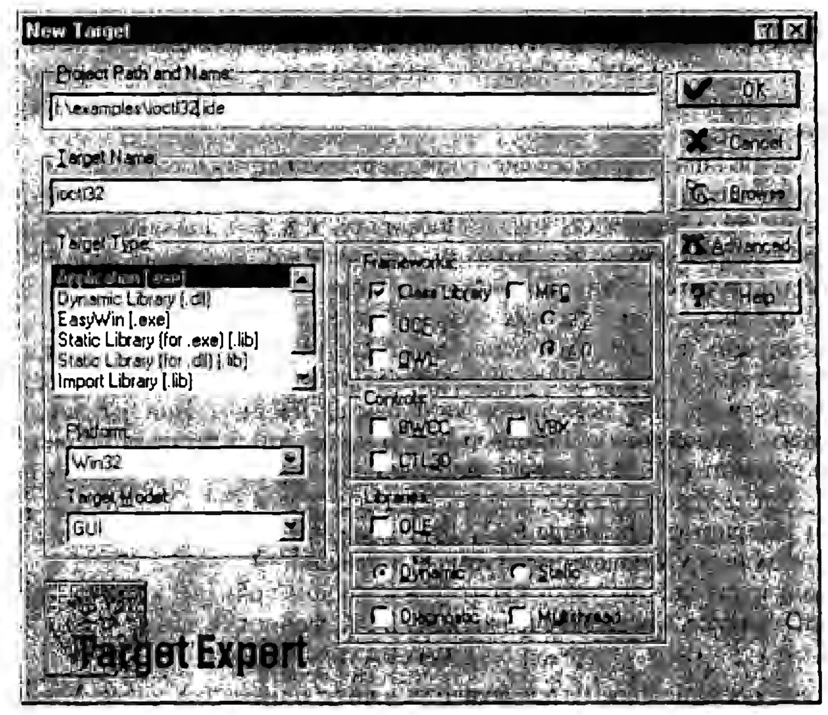


Рис. 83.2. Окно задания характеристик проекта в среде Borland C++ 5.0

Структура программы 83-01.CPP напоминает пример 82.1. В приложении не создается главное окно и, соответственно, отсутствует цикл обработки сообщений. Из изобразительных средств Windows используется только функция MessageBox(), выводящая на экран все полученные из виртуального драйвера данные (пример 83.1, продолжение).

Пример 83.1 (продолжение). IOCTL-интерфейс. Файл 83-01.CPP приложения Windows

```
#define STRICT
#include <windows.h>
#define DIOC_FUNC1 1
#define DIOC_FUNC2 2
struct DESCR{
  WORD lim;                //Граница (биты 0...15)
  WORD base_l;             //База, биты 0...15
  BYTE base_m;             //База, биты 16...23
  BYTE attr_1;             //Байт атрибутов 1
```

```
  BYTE attr_2;              //Граница (биты 16...19) и атрибуты 2
  BYTE base_h;              //База, биты 24...31
 };
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  DESCR sr[2];              //Массив из двух дескрипторов сегментов
  char txt[300];            //Строка для вывода в окно сообщения
  int InBuffer=0;           //Буфер для упаковки двух сегментов
  int OutBuffer[3];         //Буфер для вывода из драйвера трех адресов
  DWORD cb;                 //Счетчик
  char unsigned HLimCS;     //Для старших 4 бит границы сегмента
//Откроем драйвер
  HANDLE hVMyD=CreateFile("\\\\.\\VMYD",0,0,NULL,0,
        FILE_FLAG_DELETE_ON_CLOSE,NULL);
//Вызовем функцию 1 (она во входных параметрах не нуждается)
  DeviceIoControl(hVMyD,DIOC_FUNC1,NULL,0,&OutBuffer,12,&cb,0);
//Получили в буфере OutBuffer все три линейных адреса
//Подготовим параметры для вызова функции 2
  InBuffer=(DWORD)_DS;      //DS в младшей половине InBuffer
  InBuffer=InBuffer<<16;    //Сдвинем DS в старшую половину
  InBuffer|=_CS;            //Наложим CS на младшую половину
//Вызовем функцию 2
  DeviceIoControl(hVMyD,DIOC_FUNC2,&InBuffer,4,sr,16,&cb,0);
/*Получили в буфере sr оба дескриптора. Выделим для наглядности старшие 4 би-
та границ сегментов, находящиеся в байте атрибута 2*/
  HLimCS=sr[0].attr_2;
  HLimCS=HLimCS&0xF;        //Оставим только старшие 4 бита границы
//Преобразуем все результаты в символьную форму
  wsprintf(txt,"Адреса функций:\nGetVer=%lXh\nFunc1= %lXh\nFunc2= %lXh\n\n"
    "Сегмент команд:\nЛинейный адрес=%X%X%Xh"
    "\nАтрибут 1 = %Xh   Атрибут 2 = %Xh\nГраница сегмента = %X%Xh"
    "\n\nСегмент данных:\nЛинейный адрес = %X%X%Xh"
    "\nАтрибут 1 = %Xh   Атрибут 2 = %Xh\nГраница сегмента = %Xh",
    OutBuffer[0],OutBuffer[1],OutBuffer[2],
          sr[0].base_h,sr[0].base_m,sr[0].base_l,sr[0].attr_1,
          sr[0].attr_2,HLimCS,sr[0].lim,
          sr[1].base_h,sr[1].base_m,sr[1].base_l,sr[1].attr_1,
          sr[1].attr_2,sr[1].lim);
//Выведем окно сообщения с результатами
  MessageBox(NULL,txt,"Info",MB_ICONINFORMATION);
  return 0;
  }
```

В начале программы определяются константы DIOC_FUNC1 и DIOC_FUNC2, а также структура DESCR, полностью взятая из примера 76.1. В функции WinMain() вводятся необходимые переменные; их состав описывался выше. Содержательная часть программы включает вызов функции CreateFile(), открывающей драйвер и два вызова DeviceIoControl() с указанием сначала кода действия DIOC_FUNC1, а затем DIOC_FUNC2. Функция DIOC_FUNC1 не требует входных параметров, поэтому на месте адреса входного буфера в числе параметров DeviceIoControl() указан нулевой указатель NULL. Нуль указан также в качестве числа передаваемых байтов (4-й параметр функции).

Перед вызовом функции 2 драйвера необходимо подготовить исходные параметры – объединить два селектора, находящихся в регистрах DS и CS, в одно двойное слово. Для этого содержимое DS помещается в младшую часть буфера InBuffer, сдвигается в нем на 16 разрядов влево с помощью оператора Си++ << и объединяется с содержимым CS с помощью операции побитового ИЛИ (оператор |). Предложение

```
  InBuffer|=_CS;
```

представляет собой компактную запись операции побитового сложения начального содержимого InBuffer с содержимым регистра CS:

```
InBuffer=_CS | InBuffer;
```

Тридцатидвухразрядные приложения Windows работают в плоской модели памяти, и граница сегмента команд составляет FFFFFFFFh. В этом случае величина границы указывается в селекторе не в байтах, а в блоках по 4 Кбайт, о чем свидетельствует установленный бит дробности G в байте атрибутов 2 (см. рис. 66.1). Таким образом, при формировании значения границы следует учитывать не только байты 0 и 1 дескриптора, где содержатся младшие 16 бит границы, но и байт 6, где биты границы 19...16 занимают младшую половину байта.

Для выделения из байта атрибутов младшей половины в программе предусмотрена вспомогательная переменная HLimCS, в которую сначала переносится байт атрибутов 2 дескриптора сегмента команд, а затем с помощью операции побитового умножения И обнуляется ее старшая половина. Эта переменная используется в дальнейшем при выводе на экран значения границы. Для сегмента команд описанная операция не нужна, так как его граница указывается не в блоках по 4 Кбайт, а в байтах.

Оставшаяся часть программы посвящена преобразованию полученных из драйвера числовых данных в символьную форму (функция Windows wsprintf()) и выводу их на экран в виде окна сообщения (функция MessageBox()). На рис. 83.3 приведен вывод рассмотренной программы.

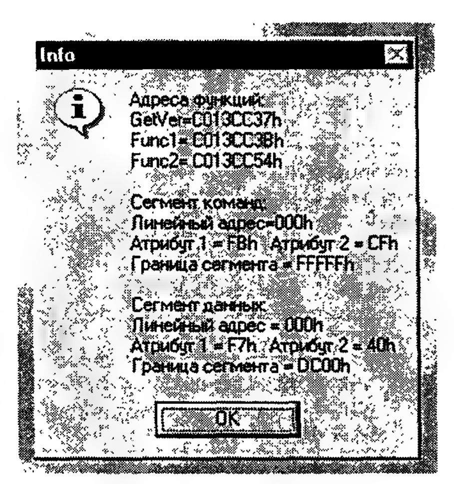


Рис. 83.3. Результат работы приложения 83-01

Статья 84. **Обращение к физической памяти в 32-разрядном приложении**

Обращение к физической памяти в 32-разрядном приложении не имеет особой специфики, за исключением способа вызова драйвера. Рассмотрим 32-разрядный вариант примера 78.1. Будем считать, что нас интересует заранее определенная область физических адресов, именно адреса F0000h...FFFFFh, занятые ПЗУ BIOS. Предусмотрим в драйвере единственную прикладную функцию, которая возвращает в приложение линейный базовый адрес этой области (пример 84.1). Поскольку 32-разрядное приложение работает в плоской модели памяти, этого адреса достаточно, чтобы обратиться из приложения к любому байту заданного диапазона физических адресов.

Пример 84.1. Обращение к физической памяти. Файл 84-01.ASM виртуального драйвера

```
.386p
.XLIST
include vmm.inc
include vwin32.inc
.LIST
DIOC_FUNC1=1
```

```
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h,Undefined_Init_Order
;=====================
VxD_REAL_INIT_SEG
...      ;Стандартная процедура инициализации реального режима
VxD_REAL_INIT_ENDS
;=====================
VxD_CODE_SEG
BeginProc VMyD_Control
Control_Dispatch W32_DeviceIOControl,IOControl
        clc
        ret
EndProc VMyD_Control
BeginProc IOControl
        cmp    dword ptr [ESI.dwIOControlCode],DIOC_GETVERSION
        je     GetVer
        cmp    dword ptr [ESI.dwIOControlCode],DIOC_FUNC1
        je     Func1
        clc
        ret
GetVer: xor    EAX,EAX
        clc
        ret
Func1:  mov    EDI,dword ptr [ESI.lpvOutBuffer]
        VMMCall _MapPhysToLinear,<0F0000h,10000h,0>
        mov    [EDI],EAX
        clc
        ret
EndProc IOControl
VxD_CODE_ENDS
end VMyD_Real_Init
```

Для получения линейного адреса, соответствующего заданному физическому, в драйвере вызывается функция VMM MapPhysToLinear. Как уже отмечалось ранее, это фактически макрос, который, при указании его с набором параметров, отправляет их в стек и вызывает соответствующую функцию VMM. В данном варианте параметры задаются в виде числовых значений, что дает возможность использовать макрос в его стандартном виде. Полученный (в регистре EAX) линейный базовый адрес заданного диапазона передается в буфер приложения.

Приложение, работающее с описанным выше драйвером, не имеет каких-либо любопытных особенностей. В нем обычным образом открывается драйвер, после чего вызовом функции DeviceIoControl в драйвер посылается код действия 1. Возвращаемый из драйвера линейный адрес поступает непосредственно в переменную OutBuffer, объявленную как указатель на символы. С помощью этого указателя приложение имеет доступ ко всему диапазону адресов BIOS. В качестве демонстрации 8 байт ПЗУ начиная с адреса FFFF5h копируются в символьный массив revision, в конец этого массива записывается нуль, служащий идентификатором конца символьной строки, и вся строка revision выводится в окно сообщений (пример 84.1, продолжение).

Пример 84.1 (продолжение). Обращение к физической памяти. Файл 84-01.CPP приложения Windows

```
#define STRICT
#include <windows.h>
#define DIOC_FUNC1 1
int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int){
  char* OutBuffer;
  char revision[9];
```

```
  DWORD cb;
//Откроем драйвер
  HANDLE hVMyD=CreateFile("\\\\.\\VMYD",0,0,NULL,0,
     FILE_FLAG_DELETE_ON_CLOSE,NULL);
//Вызовем функцию 1
  DeviceIoControl(hVMyD,DIOC_FUNC1,NULL,0,&OutBuffer,4,&cb,0);
//Извлечем из полученных данных дату выпуска BIOS
  for(int i=0;i<8;i++)
    revision[i]=OutBuffer[i+0xfff5];
  revision[8]=0;
//Выведем окно сообщения с результатами
  MessageBox(NULL,revision,"Info",MB_ICONINFORMATION);
  return 0;
  }
```

Статья 85. **Обработка аппаратных прерываний в 32-разрядном приложении**

В статье 80 были описаны общие принципы обслуживания аппаратных прерываний в защищенном режиме, а в статье 81 – особенности разработки виртуальных драйверов для этой цели. Там же отмечалось, что в 32-разрядных приложениях Windows обработка прерываний возможна только при наличии в системе виртуального драйвера для используемого уровня прерываний.

Настоящая статья посвящена виртуальным драйверам, предназначенным для обслуживания измерительной или управляющей аппаратуры автоматизированных установок, работающих в режиме прерываний, если программа управления установкой является 32-разрядным приложением Windows. Рассматриваемый ниже пример отлаживался на макете таймера-счетчика, описанного в статье 80.

В 32-разрядных приложениях Windows, так же как и в 16-разрядных, прикладная обработка аппаратных прерываний требует решения трех задач:

- подключения обработчика прерываний, входящего в состав драйвера, к требуемому уровню аппаратных прерываний;
- обмена данными между обработчиком прерываний и приложением, в частности пересылки в приложение результатов измерений;
- оповещения приложения о приходе прерывания, что наиболее эффективно достигается путем вызова из драйвера асинхронной функции приложения.

Ввиду относительной сложности этих вопросов, мы последовательно рассмотрим три варианта программ обслуживания прерываний, первый из которых будет решать лишь задачу обработки прерываний в драйвере; во втором будет введена передача результатов измерений в приложение, а в третьем будет присутствовать и асинхронная обработка прерываний в приложении. Все три варианта, по существу, будут представлять собой программные комплексы, поскольку в каждый из них будет входить как приложение Windows, так и соответствующий ему виртуальный драйвер.

Исходные тексты первого программного комплекса приведены ниже. Рассмотрим сначала приложение Windows (пример 85.1).

Пример 85.1. Обработка аппаратных прерываний. Файлы приложения Windows

Заголовочный файл 85-01.H

```
//Определения констант
#define MI_START 100
```

```
#define MI_ADDR 101
#define MI_EXIT 102
#define DIOC_INIT 1
#define DIOC_ADDR 2
//Прототипы функций
void Register(HINSTANCE);
void Create(HINSTANCE);
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
void InitCard();
void GetAddr();
```

Файл ресурсов 85-01.RC

```
#include "85-01.h"
Main MENU{
  POPUP "Режимы"  {
    MENUITEM "Пуск",MI_START
    MENUITEM "Адрес драйвера",MI_ADDR
    MENUITEM SEPARATOR
    MENUITEM "Выход",MI_EXIT
    }
  }
```

Программный файл 85-01.CPP

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include "85-01.h"
HANDLE hVMyD;                //Дескриптор открытого драйвера
char szClassName[]="MainWin";//Имя класса главного окна
char szTitle[]="Аппаратные прерывания - 1";//Заголовок окна
DWORD cbRet;                 //Длина передаваемых в приложение данных
//Главная функция WinMain
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE,LPSTR,int){
  MSG msg;
  Register(hInstance);       //Регистрация класса главного окна
  Create(hInstance);         //Создание и показ окна
  while(GetMessage(&msg,NULL,0,0))//Цикл обработки сообщений
    DispatchMessage(&msg);
  return 0;
  }
//Функция Register регистрации класса окна
void Register(HINSTANCE hInst){
  WNDCLASS wc;
  memset(&wc,0,sizeof(wc));
  wc.lpszClassName=szClassName;
  wc.hInstance=hInst;
  wc.lpfnWndProc=WndProc;
  wc.lpszMenuName="Main";  //В главном окне будет меню
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  RegisterClass(&wc);
  }
//Функция Create создания и показа окна
void Create(HINSTANCE hInst){
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
         10,10,300,150,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  }
```

```
//Оконная процедура
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);//Переход по сообщению WM_CREATE
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);//Переход по сообщению WM_COMMAND
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);//Переход по сообщению WM_DESTROY
  default:
    return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
//Функция OnCreate обработки сообщения WM_CREATE при создании окна
BOOL OnCreate(HWND,LPCREATESTRUCT){
//Откроем драйвер и получим его дескриптор
  hVMyD=CreateFile("\\\\.\\VMYD",0,0,NULL,0,FILE_FLAG_DELETE_ON_CLOSE,NULL);
  return TRUE;
  }
//Функция OnCommand обработки сообщений WM_COMMAND от пунктов меню
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id)  {
    case MI_START:
      InitCard();
      break;
    case MI_ADDR:
      GetAddr();
      break;
    case MI_EXIT:
      DestroyWindow(hwnd);
    }
  }
//Функция OnDestroy обработки сообщения WM_DESTROY о завершении приложения
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
//Функция InitCard передачи драйверу констант инициализации таймера-счетчика
void InitCard(){
  struct {               //Набор констант каналов
    short unsigned int C0;
    short unsigned int C1;
    short unsigned int C2;
  }InBuffer;
  InBuffer.C0=20;
  InBuffer.C1=50000;
  InBuffer.C2=10000;
//Вызовем процедуру драйвера DIOC_INIT
  DeviceIoControl(hVMyD,DIOC_INIT,&InBuffer,6,NULL,0,&cbRet,0);
  }
//Функция GetAddr получения из драйвера информации о его линейном адресе
//Вызовем процедуру драйвера DIOC_ADDR
void GetAddr(){
  char txt [80];
  int unsigned Addr;
  DeviceIoControl(hVMyD,DIOC_ADDR,NULL,0,&Addr,4,&cbRet,0);
  wsprintf(txt,"Базовый адрес = %Xh",Addr);
  MessageBox(NULL,txt,"Info",MB_ICONINFORMATION);
  }
```

В предыдущих примерах программный проект включал в себя максимум два файла: исходный текст программы с расширением .CPP и файл ресурсов с расширением .RC, в котором описывался состав (сценарий) меню. При этом в программный файл, кроме собственно программы, входили еще и описания прототипов используемых

функций и определения констант. Поскольку эта часть программы часто имеет значительный объем и к тому же мало изменяется от программы к программе, она выделяется в отдельный файл, называемый заголовочным и имеющий обычно расширение Н. Настоящий пример составлен именно таким образом.

В заголовочном файле 85-01.Н определены константы двух видов. Константы с префиксом MI_ являются идентификаторами пунктов меню и будут использованы как в файле ресурсов, где они сопоставляются с конкретными командами меню, так и в тексте программы для определения того, какая команда меню выбрана пользователем. Константы с префиксом DIOC_ являются кодами действий, которые программа будет пересылать в драйвер, чтобы заказать требуемое действие. По существу, эти числа являются номерами API-функций драйвера.

Вслед за определением констант в заголовочном файле приведены прототипы всех функций, использованных в приложении (кроме, естественно, системных функций, которые описаны в системных заголовочных файлах).

В последующих, более сложных вариантах программ в заголовочный файл будут вводиться дополнительные элементы (если в программу будут включаться новые функции или в меню – новые команды).

Файл ресурсов 85-01.RC начинается с оператора #include, который подключает заголовочный файл 85-01.Н. В процессе трансляции файла ресурсов из заголовочного файла будут извлекаться числовые значения констант-идентификаторов команд меню.

Сценарий меню предполагает наличие в меню трех команд: "Пуск" для инициализации таймера-счетчика и запуска процесса накопления событий, "Адрес драйвера" для вывода на экран линейного адреса первой процедуры драйвера, который можно использовать в процессе его отладки, и "Выход" для завершения программы.

Общая структура приложения Windows не отличается от рассмотренной в предыдущих примерах. После создания главного окна с полоской меню приложение входит в цикл обработки сообщений, ожидая выбора пользователем тех или иных пунктов меню. Получив из Windows идентификатор выбранной команды, приложение переходит на соответствующую прикладную функцию (OnCreate(), OnCommand() или OnDestroy()), после завершения которой возвращается в цикл обработки сообщений на ожидание следующей команды пользователя.

Сообщение WM_CREATE посылается в приложение системой Windows в процессе создания окна, чтобы программист, перехватив это сообщение, мог выполнить необходимые инициализирующие действия. В рассматриваемом примере в функции OnCreate() вызовом функции Windows CreateFile() открывается драйвер так же, как это делалось в примере 83.1. Полученный в качестве возвращаемого значения дескриптор драйвера сохраняется в глобальной переменной hVMyD для дальнейшего использования.

Приложение обращается к драйверу в двух случаях. Выбор пользователем команды меню "Адрес драйвера" приводит к вызову прикладной функции GetAddr(), которая вызовом DeviceIoControl() посылает в драйвер код действия DIOC_ADDR. В ответ драйвер через буфер вывода Addr возвращает в приложение линейный адрес своей первой процедуры VMyDControl. По этому адресу можно вычислить (по листингу трансляции драйвера) другие характерные адреса драйвера и использовать их в качестве точек останова при отладке или исследовании драйвера с помощью отладчика SoftICE.

Более содержательное обращение к драйверу выполняется в функции InitCard(), инициируемой при выборе пользователем команды меню "Пуск". Здесь в вызове

DeviceIoControl() содержится код действия DIOC_INIT и адрес входного (для драйвера) буфера InBuffer, в котором содержатся три константы C0, C1 и C2 настройки платы таймера-счетчика. Поскольку эти три 2-байтовых константы в сумме занимают 6 байт, в параметрах вызова DeviceIoControl() в качестве размера входного буфера указано число 6. В процедуре драйвера, соответствующей коду действия DIOC_INIT, выполняется инициализация и пуск таймера-счетчика. Окончание временного интервала вызывает прерывание уровня 5, которым занимается обработчик, входящий в состав драйвера. Приложение не получает информации ни об истечении временного интервала, ни о числе сосчитанных событий (хотя драйвер, как мы увидим позже, выводит результаты измерений в свое окно сообщения).

Чтение числа накопленных событий в таком варианте программы возможно только синхронным образом, с помощью еще одного запроса DeviceIoControl() с кодом действия, отличным от уже использованных. Драйвер, обрабатывая этот код действия, может передать в выходной буфер, т. е. непосредственно в приложение, результат измерений в точности так же, как передается адрес в ответ на запрос DIOC_ADDR. Однако приложение, не получая от драйвера информации о приходе прерывания, не имеет возможности узнать, когда именно в выходном буфере появится результат измерений и когда, следовательно, имеет смысл запрашивать драйвер.

Рассмотрим теперь исходный текст программы драйвера (пример 85.1, продолжение).

Пример 85.1 (продолжение). Обработка аппаратных прерываний. Файл 85-01.ASM виртуального драйвера

```
;Функции драйвера:
;0 GetAddr - Проверка версии
;1 VMyD_Init - Инициализация и пуск платы
;2 GetAddr - Передача в приложение базового адреса драйвера
DIOC_INIT=1
DIOC_ADDR=2
.386p
.XLIST
include vmm.inc
include vwin32.inc
include shell.inc
include vpicd.inc
.LIST
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h, Undefined_Init_Order
;======================
VxD_REAL_INIT_SEG
...      ;Стандартная процедура инициализации реального режима
VxD_REAL_INIT_ENDS
;=====================
VxD_DATA_SEG
Mesg db '********-VMyD message',0
result dw 0                ;Результат измерений
IRQ_Handle dd 0            ;Дескриптор виртуального прерывания
VMyD_Int13_Desc label dword;32-битовый адрес следующей далее структуры
VPICD_IRQ_Descriptor <5,,OFFSET32 VMyD_Int_13>;Структура с информацией
                           ;о виртуализованном прерывании
VxD_DATA_ENDS
;=====================
VxD_CODE_SEG
;Управляющая процедура
BeginProc VMyD_Control
Control_Dispatch W32_DeviceIOControl,IOControl;Переход по сообщению
W32_DeviceIOControl
```

```
Control_Dispatch Device_Init, VMyD_Device_Init;Переход по сообщению
Device_Init
        clc
        ret
EndProc VMyD_Control
;------------------------------
;Процедура, вызываемая при инициализации драйвера системой Windows
BeginProc VMyD_Device_Init
        mov    EDI,OFFSET32 VMyD_Int13_Desc
        VxDCall VPICD_Virtualize_IRQ
        mov    IRQ_Handle,EAX
        clc
        ret
EndProc VMyD_Device_Init
;-----------------------
;Точка входа в драйвер по вызовам DeviceIoControl
BeginProc IOControl
        cmp    dword ptr [ESI.dwIOControlCode],DIOC_GETVERSION
        je     GetVer
        cmp    dword ptr [ESI.dwIOControlCode],DIOC_INIT
        je     VMyD_Init
        cmp    dword ptr [ESI.dwIOControlCode],DIOC_ADDR
        je     GetAddr
        clc
        ret
EndProc IOControl
;---------------------------
;Процедура подтверждения поддержки драйвером IOCTL-интерфейса
BeginProc GetVer
        xor    EAX,EAX
        clc
        ret
EndProc GetVer
;---------------------------
;Процедура передачи в приложение линейного базового адреса драйвера
BeginProc GetAddr
        mov    EDI,dword ptr [ESI.lpvOutBuffer]
        mov    [EDI],offset32 VMyD_Control
        clc
        ret
EndProc GetAddr
;------------------------------
;Процедура инициализации программируемого устройства
BeginProc VMyD_Init
;Общий сброс   платы
        mov    DX,30Ch
        in     AL,DX
;Получим адрес буфера с данными настройки
        mov    EDI,dword ptr [ESI.lpvInBuffer]
;Засылаем управляющие слова по каналам
        mov    DX,303h      ;Регистр команд
        mov    AL,36h       ;Канал 0, режим 3
        out    DX,AL
        mov    AL,70h       ;Канал 1, режим 0
        out    DX,AL
        mov    AL,0B6h      ;Канал 2, режим 3
        out    DX,AL
;Программируем канал 0 - 1-ю половину таймера
        mov    AX,[EDI+0]   ;1-й параметр
        mov    DX,300h
        out    DX,AL        ;Младший байт частоты
```

```
        xchg   AL,AH
        out    DX,AL        ;Старший байт частоты
;Программируем канал 1 - 2-ю половину таймера
        mov    AX,[EDI+2]  ;2-й параметр
        mov    DX,301h
        out    DX,AL        ;Младший байт интервала
        xchg   AL,AH
        out    DX,AL        ;Старший байт интервала
;Программируем внутренний генератор
        mov    AX,[EDI+4]  ;3-й параметр
        mov    DX,302h
        out    DX,AL
        xchg   AH,AL
        out    DX,AL
;Сбрасываем счетчик
        mov    DX,308h
        out    DX,AL
;Установим флаг S2 разрешения счета
        mov    DX,30Bh
        in     AL,DX
;Размаскируем прерывания
        mov    EAX,IRQ_Handle
        VxDCall VPICD_Physically_Unmask
        clc
        ret
EndProc VMyD_Init
;---------------------------------
;Процедура обработки аппаратного прерывания
BeginProc VMyD_Int_13, High_Freq
        pushad
;Получим результат измерений
        mov    DX,309h      ;Порт старшего байта
        in     AL,DX        ;Получили старший байт результата
        mov    AH,AL        ;Отправим его в старшую половину AX
        dec    DX           ;Порт младшего байта
        in     AL,DX        ;Получили младший байт результата
        mov    result,AX    ;Сохраним весь результат в полях данных драйвера
;Сброс флагов готовности и разрешения счета
        mov    DX,30Ah
        out    DX,AL
;Выполним завершающие действия в PIC и выведем результаты
        mov    EAX,IRQ_Handle;Дескриптор IRQ
        VxDCall VPICD_Phys_EOI;Пошлем в контроллер команду EOI
        VxDCall VPICD_Physically_Mask;Замаскируем наш уровень прерываний
;Выведем полученный результат в виде сообщения драйвера
        mov    ESI,offset32 mesg
        mov    AX,result
        call   Word_ascii
        call   Info
        popad
        clc
        ret
EndProc VMyD_Int_13
;----------------------------
;Процедуры преобразования чисел в символьную форму и вывода сообщения
BeginProc Info
        ...
EndProc Info
BeginProc Word_ascii
        ...
EndProc Word_ascii
```

```
BeginProc Bin_ascii
    ...
EndProc Bin_ascii
VxD_CODE_ENDS
end VMyD_Real_Init
```

В управляющей процедуре теперь обрабатываются два сообщения – Device_Init, посылаемое в драйвер системой Windows в процессе его инициализации, и W32_DeviceIOControl, которое является следствием вызова в приложении функции IOCTL-интерфейса DeviceIoControl(). В процедуре драйвера IoControl пришедший вместе с этим сообщением код действия сравнивается с заданными, после чего управление передается на ту или иную процедуру драйвера.

В драйвере можно выделить три процедуры, имеющие отношение к обслуживанию прерываний. На этапе инициализации драйвера системой Windows (процедура VMyD_Device_Init) выполняются виртуализация запросов на прерывания уровня 5 и сохранение дескриптора виртуализованного прерывания в двухсловной ячейке IRQ_Handle.

В ответ на выбор пользователем команды меню "Пуск" (процедура VMyD_Init) после приема из приложения констант настройки и инициализации таймера-счетчика выполняется размаскирование уровня 5 в контроллере прерываний вызовом функции виртуального контроллера прерываний VPICD_Physically_Unmask.

Наконец, в обработчике прерываний (процедура VMyD_Int_13) после получения из счетчика результата измерений, выполняются обычные завершающие действия – посылка в контроллер команды EOI и маскирование уровня 5 контроллера прерываний. В конце этой процедуры выполняется формирование и вывод на экран сообщения драйвера с результатом измерений.

В целом можно сказать, что приведенный выше текст отличается от программы драйвера для 16-разрядного приложения из примера 81.1 лишь средствами связи с приложением.

Статья 86. **Аппаратные прерывания и передача данных в 32-разрядном приложении**

Введем в рассмотренную программу средства передачи данных из обработчика аппаратных прерываний в приложение. Казалось бы, что, если заранее с помощью вызова DeviceIoControl() передать драйверу линейный адрес буфера вывода в программу, в обработчике прерываний можно записать в этот буфер любые данные, в частности результаты измерений. Однако обработчик аппаратных прерываний активизируется асинхронно, и это накладывает серьезные ограничения на его возможности. Вообще любая асинхронная программа выполняется, как говорят, в другом контексте, нежели прерванная ею синхронная программа. Линейные адреса полей данных или процедур приложения, полученные драйвером на синхронном уровне, недействительны в то время, когда драйвер выполняет асинхронную процедуру. Поэтому пересылка данных из обработчика прерываний в приложение, хотя и возможна, но требует выполнения некоторых дополнительных действий как в приложении, так и (в основном) в драйвере. Рассмотрим сначала изменения, вносимые в приложение.

Предусмотрим среди глобальных данных приложения переменную Data, в которую драйвер сможет переслать результаты измерения:

```
short unsigned int Data=1234h;//Начальное значение для отладки
```

Для того чтобы драйвер мог обращаться к этой ячейке, необходимо переслать в драйвер ее адрес. Это удобно выполнить в функции инициализации InitCard(), включив указатель на беззнаковое короткое целое в передаваемую в драйвер структуру InBuffer и инициализировав этот дополнительный член структуры адресом переменной Data. Функция InitCard() примет теперь следующий вид:

```
void InitCard(){
  struct {
    short unsigned int C0;
    short unsigned int C1;
    short unsigned int C2;
    short unsigned int* dptr;
    }InBuffer;
  InBuffer.C0=20;           //Канал 0
  InBuffer.C1=50000;        //Канал 1
  InBuffer.C2=10000;        //Канал 2
  InBuffer.dptr=&Data;      //Адрес переменной Data
//Вызовем процедуру драйвера DIOC_INIT
  DeviceIoControl(hVMyD,DIOC_INIT,&InBuffer,8,NULL,0,&cbRet,0);
  }
```

Обратите внимание на изменение параметра, определяющего размер входного буфера: добавление в структуру InBuffer адреса переменной Data увеличило его длину до 8 байт.

Для чтения значения переменной Data после передачи в нее из драйвера результата измерений предусмотрим в меню команду "Чтение данных". Для этого в файл заголовков следует включить определение идентификатора этой команды

```
#define MI_DATA 103
```

а в файл ресурсов – предложение

```
MENUITEM "Чтение данных",MI_DATA
```

Функция обработки сообщений от меню дополнится анализом на выбор команды "Чтение данных" с идентификатором MI_DATA и примет вид

```
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id)  {
    case MI_START:
      InitCard();
      break;
    case MI_ADDR:
      GetAddr();
      break;
    case MI_DATA:
      ReadData();
      break;
    case MI_EXIT:
      DestroyWindow(hwnd);
    }
  }
```

Новая функция ReadData() преобразует значение переменной Data в символьную форму и выводит его в окно сообщения в виде 16-ричного и десятичного числа (такой вывод может пригодиться в процессе отладки драйвера, когда контролировать результат удобнее в 16-ричной форме):

```
void ReadData(){
  char txt[80];
  wsprintf(txt,"Результат = %Xh = %u",Data,Data);
```

```
MessageBox(NULL,txt,"Info",MB_ICONINFORMATION);
  }
```

Как мы увидим позже, драйвер для получения доступа к переменной приложения должен будет выполнить блокирование принадлежащей ей памяти. Перед завершением программы эту память необходимо разблокировать, для чего нужно вызвать соответствующую прикладную функцию драйвера. Этот вызов естественно включить в процедуру обработки сообщения WM_DESTROY:

```
void OnDestroy(HWND){
  DeviceIoControl(hVMyD,DIOC_EXIT,NULL,0,NULL,0,&cbRet,0);
  PostQuitMessage(0);
  }
```

Вызов не требует передачи каких-либо параметров, поэтому и адреса и размеры обоих буферов обмена информацией с драйвером имеют нулевые значения. Определение кода действия DIOC_EXIT должно быть включено в заголовочный файл (а также и в программу драйвера):

```
#define DIOC_EXIT 3
```

Таким образом, изменения в приложении носят скорее организационный, чем содержательный характер. Фактически вся работа по получению доступа к переменной приложения ложится на драйвер (пример 86.1).

Пример 86.1. Доступ к данным приложения из обработчика аппаратных прерываний. Файл 86-01.ASM виртуального драйвера

```
;Функции драйвера:
;0 Проверка версии
;1 VMyD_Init - Инициализация и пуск платы; блокирование физической памяти
;2 GetAddr - Передача в приложение базового адреса драйвера
;3 VMyD_Exit - Разблокирование физической памяти
DIOC_INIT=1
DIOC_ADDR=2
DIOC_EXIT=3
.386p
.XLIST
include vmm.inc
include vwin32.inc
include vpicd.inc
.LIST
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h,Undefined_Init_Order
;======================
VxD_REAL_INIT_SEG
...      ;Инициализация реального режима
VxD_REAL_INIT_ENDS
;=====================
VxD_DATA_SEG
result dw 0              ;Результат измерений
IRQ_Handle dd 0          ;Дескриптор виртуального прерывания
VMyD_Int13_Desc label dword;32-битовый адрес следующей далее структуры
VPICD_IRQ_Descriptor <5,,OFFSET32 VMyD_Int_13>
lpdata dd 0              ;Адрес переменной Data приложения
ndata dd 0               ;Число блокируемых страниц
VxD_DATA_ENDS
;=====================
VxD_CODE_SEG
BeginProc VMyD_Control
Control_Dispatch W32_DeviceIOControl,IOControl
Control_Dispatch Device_Init, VMyD_Device_Init
```

```
        clc
        ret
EndProc VMyD_Control
;-----------------------
BeginProc VMyD_Device_Init
        mov   EDI,OFFSET32 VMyD_Int13_Desc
        VxDCall VPICD_Virtualize_IRQ
        mov   IRQ_Handle,EAX
        clc
        ret
EndProc VMyD_Device_Init
;-----------------------
BeginProc IOControl
        cmp   dword ptr [ESI.dwIOControlCode],DIOC_GETVERSION
        je    GetVer
        cmp   dword ptr [ESI.dwIOControlCode],DIOC_INIT
        je    VMyD_Init
        cmp   dword ptr [ESI.dwIOControlCode],DIOC_ADDR
        je    GetAddr
        cmp   dword ptr [ESI.dwIOControlCode],DIOC_EXIT
        je    VMyD_Exit
        clc
        ret
EndProc IOControl
;--------------------------
BeginProc GetVer
        xor   EAX,EAX
        clc
        ret
EndProc GetVer
;--------------------------
BeginProc GetAddr
        mov   EDI,dword ptr [ESI.lpvOutBuffer]
        mov   [EDI],offset32 VMyD_Control
        clc
        ret
EndProc GetAddr
;-----------------------------
BeginProc VMyD_Init
;Общий сброс  платы
        mov   DX,30Ch      ;(1)
        in    AL,DX        ;(2)
;Получим адрес буфера приложения и заблокируем физическую память переменной
Data
        mov   EDI,dword ptr [ESI.lpvInBuffer];(3)Адрес буфера приложения
        mov   EAX,[EDI+6] ;(4)Получим адрес переменной Data приложения
        and   EAX,0FFFh   ;(5)Выделим младшие 12 бит - смещение на странице
        mov   EBX,EAX     ;(6)Сохраним его в EBX
        mov   EAX,[EDI+6] ;(7)Получим адрес переменной Data приложения
        mov   ECX,EAX     ;(8)Занесем его и в ECX
        shr   ECX,12      ;(9)Полный номер начальной физической страницы
        add   EAX,1       ;(10)Прибавим длину данных - 1 к адресу переменной
        shr   EAX,12      ;(11)Полный номер конечной физической страницы
        sub   EAX,ECX     ;(12)Число страниц, требуемых для данных, - 1
        inc   EAX         ;(13)Число страниц, требуемых для данных
        mov   ndata,EAX   ;(14)Сохраним его
        push  PAGEMAPGLOBAL;(15)Вид блокирования
        push  EAX         ;(16)Число блокируемых страниц
        push  ECX         ;(17)Полный номер начальной блокируемой страницы
        VMMCall _LinPageLock;(18)EAX=новый линейный адрес страницы
        add   EAX,EBX     ;(19)Новый линейный адрес данного
```

```
        mov    lpdata,EAX  ;(20)Сохраним его
        add    ESP,12      ;(21)Восстановление стека
;Засылаем управляющие слова по каналам, сбрасываем счетчик,
;устанавливаем флаг S2 разрешения счета
;       ...    ;Эта часть программы драйвера полностью повторяет пример 85.1
;Размаскируем прерывания
        mov    EAX,IRQ_Handle
        VxDCall VPICD_Physically_Unmask
        clc
        ret
EndProc VMyD_Init
;-------------------------
BeginProc VMyD_Exit
;Разблокируем данные приложения
        mov    EAX,lpdata  ;Возьмем новый линейный адрес начальной страницы
        shr    EAX,12      ;Полный номер начальной страницы
        push   PAGEMAPGLOBAL;Тип разблокирования
        push   ndata       ;Число разблокируемых страниц
        push   EAX         ;Полный номер начальной страницы
        VMMCall _LinPageUnLock
        add    ESP,12      ;Восстановим стек
        clc
        ret
EndProc VMyD_Exit
;--------------------------------
BeginProc VMyD_Int_13, High_Freq
        pushad
;Получим результат измерений
        mov    DX,309h
        in     AL,DX
        mov    AH,AL
        dec    DX
        in     AL,DX
        mov    result,AX   ;Сохраним результат в памяти
;Сброс флагов готовности и разрешения счета
        mov    DX,30Ah
        out    DX,AL
;Выполним завершающие действия в PIC и выведем результат
        mov    EAX,IRQ_Handle
        VxDCall VPICD_Phys_EOI
        VxDCall VPICD_Physically_Mask
        mov    AX,result   ;Результат измерений
        mov    EDI,lpdata  ;Передадим данное
        mov    [EDI],AX    ;в буфер приложения
        popad
        clc
        ret
EndProc VMyD_Int_13
VxD_CODE_ENDS
end VMyD_Real_Init
```

В начале текста драйвера определяется значение добавленного кода действия DIOC_EXIT. Соответственно в процедуру IOControl включены предложения проверки пришедшего из приложения кода действия на значение DIOC_EXIT и перехода в случае равенства на процедуру VMyD_Exit.

Сущность изменений, внесенных в программу драйвера, заключается в том, что в процедуре VMyD_Init, помимо инициализации платы, осуществляется блокирование физической памяти с теми данными, к которым нужно получить доступ, а в новой процедуре VMyD_Exit – разблокирование этой памяти. Блокирование осуществляется

целыми страницами (4 Кбайт) с помощью функции VMM _LinPageLock; функция в случае своего успешного выполнения возвращает новый линейный адрес первой заблокированной страницы, посредством которого драйвер получает доступ к полям данных приложения.

Рассмотрим подготовку параметров для функции _LinPageLock в процедуре VMyD_Init нашего драйвера (для удобства ссылок предложения этой процедуры пронумерованы). В предложении 3 из структуры DIOCParams извлекается и помещается в регистр EDI адрес входного буфера. В предложении 4 линейный адрес интересующей нас переменной приложения Data забирается из входного буфера (где он располагается в байтах 6…7 от начала буфера) и помещается в регистр – EAX. Как известно, в линейном адресе биты 22...31 определяют номер элемента в каталоге страниц, т. е., в сущности, номер таблицы страниц, биты 12...21 – номер страницы, а биты 0...11 – смещение адресуемого данного на странице (рис. 86.1).

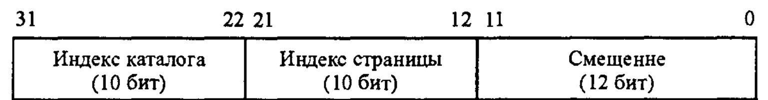


Рис. 86.1. Составляющие линейного адреса

Функция _LinPageLock требует в качестве параметра часть линейного адреса, расположенную в битах 12...31, которую можно условно рассматривать как полный номер физической страницы (не ее адрес!). В предложении 5 из линейного адреса выделяются его младшие 12 бит – смещение на логической (и физической) страницах, а в предложении 6 оно сохраняется для дальнейшего использования в регистре EBX. Далее формируются полные номера физических страниц с данными – начальной (в регистре ECX) и конечной (в регистре EAX). В предложении 7 в регистре EAX восстанавливается линейный адрес Data, а в следующем предложении он помещается еще и в ECX. Сдвигом вправо на 12 бит содержимое ECX преобразуется в полный номер (начальной) страницы. В предложении 9 к линейному адресу данного в регистре EAX прибавляется величина, на единицу меньшая, чем полная длина данных (в нашем случае всего 1 байт), что дает линейный адрес последнего байта блокируемых данных. Эта величина также сдвигается вправо на 12 бит с образованием полного номера (конечной) страницы. Разность двух номеров страниц (предложение 12) после прибавления единицы (предложение 13) дает число страниц, подлежащих блокированию. Необходимость в столь сложной процедуре определения числа блокируемых страниц объясняется тем, что блокируемые данные, даже при небольшой длине, могут начинаться на одной физической странице, а заканчиваться на другой. Полученное число страниц, подлежащих блокированию, сохраняется в ячейке ndata с целью использования в процедуре разблокирования.

В предложениях 15...17 в стек проталкиваются параметры функции VMM _LinPageLock: константа PAGEMAPGLOBAL, используемая в случае блокирования страниц ради доступа к ним из другого контекста, т. е. из асинхронной операции; число блокируемых страниц и полный номер начальной блокируемой страницы. Функция _LinPageLock в случае своего успешного выполнения возвращает в регистре EAX новый линейный адрес заблокированного участка памяти. Однако этот адрес характеризует начало страницы; наши данные располагались на странице с некоторым смещением, которое мы выделили и сохранили в регистре EBX. Теперь это смещение можно

прибавить к линейному адресу начальной страницы, чтобы получить точный адрес начала данных (предложение 19). В двух последующих предложениях полученный адрес сохраняется в предусмотренной для этого ячейке lpdat и восстанавливается состояние указателя стека, каким оно было перед помещением в стек трех параметров функции _LinPageLock.

Далее в процедуре VMyD_Init, как и в предыдущем примере, таймер-счетчик инициализируется константами, полученными из приложения через входной буфер, и размаскируется наш уровень прерываний.

В обработчике прерываний VMyD_Int_13 выполняются те же действия, что и в предыдущем примере (получение из счетчика результата измерений, сброс флагов готовности и разрешения счета, посылка в контроллер прерываний команды EOI), после чего результат измерений передается в приложение по новому "адресу-псевдониму", сохраненному ранее в ячейке lpdata.

Как уже отмечалось ранее, заблокированную память необходимо разблокировать. Это действие выполняется с помощью функции VMM _LinPageUnLock при завершении приложения, когда в драйвер посылается запрос на операцию ввода-вывода с кодом действия DIOC_EXIT. В этом случае в драйвере вызывается процедура VMyD_Exit, в которой еще раз выполняется процедура определения полного номера начальной страницы и числа заблокированных страниц. Поскольку блокирование осуществляется целыми страницами, смещение данных нас в этой процедуре уже не интересует. При разблокировании номера страниц определяются исходя из полученного в операции блокирования адреса-псевдонима.

Статья 87. **Синхронизация обработчиков прерываний в 32-разрядном приложении**

Описанная методика далека от совершенства. Она годится лишь для таких систем реального времени, в которых все действия по обработке прерываний могут быть целиком возложены на обработчик прерываний. Например, получив сигнал об изменении состояния установки, обработчик прерываний путем посылки соответствующих команд в порты установки переключает ее управляющие элементы. Основная программа в этом процессе может не участвовать и даже не знать о поступлении сигнала прерывания.

Чаще сигнал прерывания свидетельствует о наступлении такого события, о котором должна быть извещена основная программа. Типичной является ситуация, когда прерывание сигнализирует об окончании процесса измерений либо о регистрации исследуемого явления. В этом случае программа должна принять из установки накопленные данные и приступить к их обработке или визуальному представлению.

Рассмотрим дальнейшее видоизменение программного комплекса по управлению таймером-счетчиком, в который теперь введена асинхронная обработка прерываний не только в виртуальном драйвере, но и в самой программе. Учитывая относительную сложность этой методики, ниже приводятся все исходные тексты комплекса, хотя фактически в них, по сравнению с предыдущими примерами, внесены лишь некоторые дополнения (пример 87.1).

Пример 87.1. Асинхронная обработка аппаратного прерывания в приложении Windows. Тексты исходных файлов, входящих в приложение Windows

Заголовочный файл 87-01.H

```
//Определения констант
#define MI_START 100
#define MI_ADDR 101
#define MI_EXIT 102
#define MI_DATA 103
#define DIOC_INIT 1
#define DIOC_ADDR 2
#define DIOC_EXIT 3
//Прототипы функций
void Register(HINSTANCE);
void Create(HINSTANCE);
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
void InitCard();
void GetAddr();
void ReadData();
DWORD WINAPI Isr(LPVOID);
```

Файл ресурсов 87-01.RC

```
#include "87-01.h"
Main MENU{
  POPUP "Режимы"  {
    MENUITEM "Пуск",MI_START
    MENUITEM "Адрес драйвера",MI_ADDR
    MENUITEM "Чтение данных",MI_DATA
    MENUITEM SEPARATOR
    MENUITEM "Выход",MI_EXIT
    }
  }
```

Программный файл 87-01.CPP

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include "87-01.h"
char szClassName[]="MainWin";
char szTitle[]="Аппаратные прерывания - 3";
HANDLE hVMyD,hEvent,hThread;
unsigned short int Data=0x1234;
OVERLAPPED ovlp;
DWORD dwThreadID,cbRet;
HINSTANCE hInst;
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE,LPSTR,int){
  MSG msg;
  Register(hInstance);
  hInst=hInstance;
  Create(hInstance);
  while(GetMessage(&msg,NULL,0,0))
    DispatchMessage(&msg);
  return 0;
  }
void Register(HINSTANCE hInst){
  WNDCLASS wc;
  memset(&wc,0,sizeof(wc));
  wc.lpszClassName=szClassName;
  wc.hInstance=hInst;
  wc.lpfnWndProc=WndProc;
  wc.lpszMenuName="Main";
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
```

```
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  RegisterClass(&wc);
}
void Create(HINSTANCE hInst){
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
    10,10,250,140,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
}
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
  default:
    return(DefWindowProc(hwnd,msg,wParam,lParam));
  }
}
BOOL OnCreate(HWND,LPCREATESTRUCT){
  hVMyD=CreateFile("\\\\.\\VMYD",0,0,NULL,0,
     FILE_FLAG_DELETE_ON_CLOSE|FILE_FLAG_OVERLAPPED,NULL);
  return TRUE;
  }
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id)  {
    case MI_START:
      InitCard();
      break;
    case MI_DATA:
      ReadData();
      break;
    case MI_ADDR:
      GetAddr();
      break;
    case MI_EXIT:
      DestroyWindow(hwnd);
    }
  }
void OnDestroy(HWND){
  DeviceIoControl
    (hVMyD,DIOC_EXIT,NULL,0,NULL,0,&cbRet,0);
  PostQuitMessage(0);
  }
void InitCard(){
  struct {
    short unsigned int C0;
    short unsigned int C1;
    short unsigned int C2;
    unsigned short int* dptr;
  }InBuf;
  InBuf.C0=20;             //Канал 0
  InBuf.C1=50000;          //Канал 1
  InBuf.C2=10000;          //Канал 2
  InBuf.dptr=&Data;        //Адрес переменной Data
  hEvent=CreateEvent(NULL,FALSE,FALSE,NULL);//Создадим событие
  ovlp.hEvent=hEvent;
  hThread=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Isr,
               NULL,0,&dwThreadID);
  DeviceIoControl
    (hVMyD,DIOC_INIT,&InBuf,10,NULL,0,&cbRet,&ovlp);
  }
void ReadData(){
  char txt [80];
  wsprintf(txt,"Отсчет = %d = %xh",Data,Data);
  MessageBox(NULL,txt,"Info",MB_ICONINFORMATION);
```

```
void GetAddr(){
  char txt [80];
  int unsigned Addr;
  DeviceIoControl
    (hVMyD,DIOC_ADDR,NULL,0,&Addr,4,&cbRet,0);
  wsprintf(txt,"Базовый адрес = %Xh",Addr);
  MessageBox(NULL,txt,"Info",MB_ICONINFORMATION);
  }
/*Асинхронная функция, вызываемая драйвером
из обработчика аппаратного прерывания*/
DWORD WINAPI Isr(LPVOID){
  char txt [80];
  GetOverlappedResult(hVMyD,&ovlp,&cbRet,TRUE);
  wsprintf(txt,"Время истекло!\nData = %d = %xh",Data,Data);
  MessageBox(NULL,txt,"Асинхронный вызов",MB_ICONINFORMATION);
  return 0;
  }
```

В текст приложения внесено не так уж много изменений, хотя они носят весьма принципиальный характер. В полях данных появился новый дескриптор hEvent типа HANDLE для описания события, а также переменная dwThreadID, в которую система вернет идентификатор созданного потока. И событие и поток понадобятся для включения в программу асинхронной функции (она у нас названа Isr), которая будет активизироваться из обработчика аппаратного прерывания, включенного в состав виртуального драйвера. Соответственно в заголовочном файле 87-01.H описан ее прототип

```
DWORD WINAPI Isr(LPVOID);
```

В полях данных приложения объявлена также структурная переменная ovlp типа OVERLAPPED. Эта структура содержит информацию, используемую при выполнении асинхронных операций. Заметим, что слово overlapped ("с перекрытием") в данном контексте обозначает именно асинхронные операции; мы еще столкнемся с ним в других местах программы.

Структура OVERLAPPED описана в файле WINBASE.H следующим образом:

```
typedef struct _OVERLAPPED {
    DWORD   Internal;       //Системное состояние
    DWORD   InternalHigh;   //Длина передаваемых данных
    DWORD   Offset;         //Позиция в файле
    DWORD   OffsetHigh;     //Старшее слово позиции в файле
    HANDLE  hEvent;         //Дескриптор события, служащего
                            //для синхронизации потоков
} OVERLAPPED, *LPOVERLAPPED;
```

Из комментариев можно сделать заключение, что структура OVERLAPPED предназначена главным образом для асинхронных файловых операций; в нашем случае асинхронность возникает по иной причине, и в этой структуре нам понадобятся только два члена – первый и последний.

Для того чтобы виртуальный драйвер мог выполнять асинхронные операции, при его открытии функцией CreateFile() необходимо указать, кроме флага FILE_FLAG _DELETE_ON_CLOSE, еще и флаг FILE_FLAG_OVERLAPPED:

```
hVMyD=CreateFile("\\\\.\\VMYD",0,0,NULL,0,
     FILE_FLAG_DELETE_ON_CLOSE|FILE_FLAG_OVERLAPPED,NULL);
```

Синхронизация приложения Windows и драйвера (точнее, его обработчика прерываний) осуществляется с помощью двух фундаментальных понятий 32-разрядной среды (будем называть ее для краткости Win32) – потока и события. Экземпляр загру-

женной в память программы представляет собой, в терминологии Win32, процесс. Процесс не является активным объектом – ему просто принадлежит 4-гигабайтовое адресное пространство, а также другие ресурсы, в частности файлы, с которыми работает программа. Для выполнения программы следует создать в рамках процесса по крайней мере один поток. Первичный поток, который представляет собой последовательность выполнения предложений программы, система всегда сама создает при инициализации процесса. В программе с одним потоком ход выполнения программы определяется последовательностью ее предложений; все процессорное время отдается единственному потоку (разумеется, если в системе запущено несколько программ, процессорное время будет разделяться между ними).

При необходимости организовать параллельные вычисления программист может создать в рамках процесса несколько потоков. Тогда, если один из потоков, например, ждет ввода с клавиатуры, процессорное время будет отдано другому потоку, который может выполнять математическую обработку имеющихся данных или другие независимые операции.

Наличие в процессе нескольких потоков требует их взаимной синхронизации. Для этого в Win32 предусмотрен целый ряд синхронизирующих объектов: критические секции, мьютексы, события и др. Для наших целей удобно воспользоваться событием.

Событие, как и другие синхронизирующие объекты, может находиться в одном из двух состояний: свободном (signaled) и занятом (nonsignaled). Поток же, с другой стороны, можно остановить в ожидании освобождения конкретного события. Если это событие занято, поток спит и система не выделяет ему процессорного времени. При этом система знает, какое именно событие может разбудить поток. Как только это событие перейдет в свободное состояние, поток просыпается и начинает свое выполнение.

В нашем примере создание потока и синхронизирующего события выполняется в функции InitCard(), активизируемой выбором пункта "Пуск" главного меню приложения. Сначала с помощью функции CreateEvent() создается событие:

```
hEvent=CreateEvent(NULL,FALSE,FALSE,NULL);//Создадим событие
```

Первый параметр этой функции представляет собой адрес структуры защиты, позволяющей ограничить доступ к объекту. Защита объектов применяется главным образом в многопользовательских системах и нас интересовать не будет.

Второй параметр позволяет задать тип события: со сбросом вручную (тогда этот параметр должен иметь значение TRUE) или с автосбросом (FALSE). Тип сброса определяет поведение события после освобождения синхронизируемых им потоков: при автосбросе событие автоматически переводится в занятое состояние, при сбросе вручную для этого надо использовать функцию ResetEvent(). Сброс вручную применяется в основном при синхронизации одним событием нескольких потоков; в нашем случае синхронизируется один поток и мы задаем режим автосброса.

Третий параметр является флагом начального состояния события. Создаваемый нами дополнительный поток должен находиться в "спящем" состоянии до прихода прерывания, поэтому исходное состояние синхронизирующего его события должно быть занятым, чему соответствует значение FALSE.

Наконец, последний параметр определяет имя события, без которого в данном случае можно обойтись.

Функция CreateEvent() в случае успешного выполнения возвращает дескриптор созданного события. Его необходимо заслать в последний элемент структуры OVERLAPPED:

```
ovlp.hEvent=hEvent;
```

Создав событие и назначив ему занятое состояние (т. е. состояние, которое будет блокировать выполнение связанного с этим событием потока), можно создать сам поток. Фактически потоком будет являться асинхронная процедура Isr(), входящая в состав нашего приложения. Поток создастся функцией CreateThread():

```
hThread=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Isr,
                     NULL,0,&dwThreadID);
```

Нулевые значения двух первых параметров этой функции определяют использование значений по умолчанию для атрибутов защиты потока и размера его стека. Третий параметр самый важный: он определяет адрес той функции, которая будет выполняться, когда поток будет активизирован. Этот адрес представляет собой переменную типа LPTHREAD_START_ROUTINE; использование имени типа в скобках перед именем функции преобразует ее адрес в требуемый тип. Четвертый параметр передается создаваемому потоку, который может его использовать в качестве инициализирующего значения. Указание в качестве пятого параметра константы CREATE_SUSPENDED позволяет задержать начало выполнения создаваемого потока до выполнения функции ResumeThread(); нулевое значение этого параметра определяет немедленное исполнение потока (конкретно – функции Isr()). В качестве последнего параметра указывается адрес переменной, в которую функция CreateThread() вернет идентификатор нового потока.

Итак, сразу после создания потока начинает выполняться функция Isr(). Нам же надо, чтобы она активизировалась лишь в случае прихода в виртуальный драйвер аппаратного прерывания. Для перевода потока с функцией Isr() в "спящее" состояние выполняется вызов

```
GetOverlappedResult(hVMyD,&ovlp,&cbRet,TRUE);
```

При указании в качестве последнего параметра этой функции значения TRUE, функция приостанавливает выполнение потока в ожидании установки в свободное состояние того события, дескриптор которого находится в структурной переменной типа OVERLAPPED. Адрес этой переменной указывается в качестве второго параметра функции GetOverlappedResult(). Первым параметром функции выступает дескриптор драйвера, участвующего в асинхронной операции, а третьим – адрес счетчика числа передаваемых в асинхронной операции байтов (у нас передача байтов отсутствует). Таким образом, запустив поток, мы тут же его остановили в ожидании установки события.

Вернемся к функции InitCard(). Завершающим действием в этой функции после создания события и потока является обращение к драйверу с целью, во-первых, передачи ему констант инициализации платы и адреса переменной Data и, во-вторых, активизации механизма асинхронных операций. Последнее достигается указанием в качестве последнего параметра функции DeviceIoControl() адреса структуры OVERLAPPED. Этот адрес помещается в элемент lpoOverlapped структуры DIOCParams, передаваемой драйверу вместе с сообщением W32_DeviceIOControl, и может быть извлечен оттуда драйвером так же, как извлекаются из этой структуры передаваемые в драйвер прикладные данные.

На рис. 87.1 приведено главное окно приложсния 87-01 с открытым меню. Результаты выполнения приложения будут проиллюстрированы позже, после рассмотрения программы соответствующего ему виртуального драйвера (пример 87.1, продолжсние).

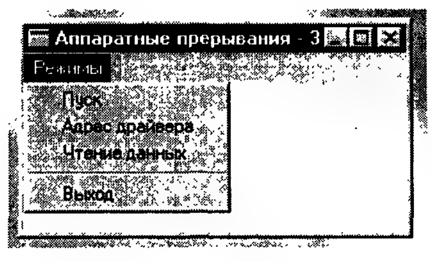


Рис. 87.1. Главное окно и меню приложения 87-01

Пример 87.1 (продолжение). Асинхронная обработка аппаратного прерывания в приложении Windows. Исходный текст виртуального драйвера 87-01.ASM

```
;Функции драйвера:
;0 Проверка версии
;1 VMyD_Init - Инициализация и пуск платы
;2 VMyD_Addr - Передача в приложение базового адреса драйвера
;3 VMyD_Exit - Разблокировка физической памяти
DIOC_INIT=1
DIOC_ADDR=2
DIOC_EXIT=3
.386p
.XLIST
include vmm.inc
include vwin32.inc
include shell.inc
include vpicd.inc
.LIST
Declare_Virtual_Device VMyD,1,0,VMyD_Control,8000h, \
     Undefined_Init_Order
;========================
VxD_REAL_INIT_SEG
BeginProc VMyD_Real_Init
        mov    AH,09h
        mov    DX,offset msg
        int    21h
        mov    AX, Device_Load_Ok
        xor    BX,BX
        xor    SI,SI
        xor    EDX,EDX
        ret
msg db 'Загружен виртуальный драйвер$'
EndProc   VMyD_Real_Init
VxD_REAL_INIT_ENDS
;=======================
VxD_DATA_SEG
result  dw 0
IRQ_Handle dd 0
VMyD_Int13_Desc label dword
VPICD_IRQ_Descriptor <5,,OFFSET32 VMyD_Int_13>
lpdata dd 0                 ;Линейный адрес данного из приложения
ndata dd 0                  ;Число блокируемых страниц с данными
lpovlp dd 0                 ;Линейный адрес структуры ovlp из приложения
novlp dd 0                  ;Число блокируемых страниц с ovlp
VxD_DATA_ENDS
;=======================
VxD_CODE_SEG
BeginProc VMyD_Control
Control_Dispatch W32_DeviceIOControl,IOControl
Control_Dispatch Device_Init, VMyD_Device_Init
```

```
        ---
        ret
EndProc VMyD_Control
;------------------------------
BeginProc VMyD_Device_Init
        mov     EDI,OFFSET32 VMyD_Int13_Desc
        VxDCall VPICD_Virtualize_IRQ
        mov     IRQ_Handle,EAX
        clc
        ret
EndProc VMyD_Device_Init
;------------------------
BeginProc IOControl
        cmp     dword ptr [ESI.dwIOControlCode],DIOC_GETVERSION
        je      GetVer
        cmp     dword ptr [ESI.dwIOControlCode],DIOC_INIT
        je      VMyD_Init
        cmp     dword ptr [ESI.dwIOControlCode],DIOC_ADDR
        je      GetAddr
        cmp     dword ptr [ESI.dwIOControlCode],DIOC_EXIT
        je      VMyD_Exit
        clc
        ret
EndProc IOControl
;--------------------------
BeginProc GetVer
        xor     EAX,EAX
        clc
        ret
EndProc GetVer
;--------------------------
BeginProc GetAddr
        mov     EDI,dword ptr [ESI.lpvOutBuffer]
        mov     [EDI],offset32 VMyD_Control
        clc
        ret
EndProc GetAddr
;------------------------------
BeginProc VMyD_Init
;Общий сброс   платы
        mov     DX,30Ch
        in      AL,DX
;Заблокируем данные Data приложения
        mov     EDI,dword ptr [ESI.lpvInBuffer];Адрес буфера приложения
        mov     EAX,[EDI+6] ;Получим адрес переменной Data приложения
        and     EAX,0FFFh   ;Выделим младшие 12 бит - смещение на странице
        mov     EBX,EAX     ;Сохраним его в EBX
        mov     EAX,[EDI+6] ;Получим адрес переменной Data приложения
        mov     ECX,EAX     ;Занесем его и в ECX
        shr     ECX,12      ;Полный номер начальной физической страницы
        add     EAX,1       ;Прибавим длину данных - 1 к адресу переменной
        shr     EAX,12      ;Полный номер конечной физической страницы с данными
        sub     EAX,ECX     ;Число страниц, требуемых для данных, - 1
        inc     EAX         ;Число страниц, требуемых для данных
        mov     ndata,EAX   ;Сохраним его
        push    PAGEMAPGLOBAL;Вид блокирования
        push    EAX         ;Число блокируемых страниц
        push    ECX         ;Полный номер начальной блокируемой страницы
        VMMCall _LinPageLock;EAX=новый линейный адрес страницы
        add     EAX,EBX     ;Новый линейный адрес данного
        mov     lpdata,EAX  ;Сохраним его
        add     ESP,12      ;Восстановление стека
;Заблокируем структуру ovlp в приложении
        mov     EAX,dword ptr [ESI.lpoOverlapped];Получим адрес структуры ovlp
```

```
        and     EAX,0FFFh   ;Выделим младшие 12 бит - смещение на странице
        mov     EBX,EAX     ;Сохраним его в EBX
        mov     EAX,dword ptr [ESI.lpoOverlapped];Получим адрес структуры ovlp
        mov     ECX,EAX     ;Занесем его и в ECX
        shr     ECX,12      ;Полный номер начальной физической страницы
        add     EAX,19      ;Прибавим длину блокируемых данных - 1
        shr     EAX,12      ;Полный номер конечной физической страницы с ovlp
        sub     EAX,ECX     ;Число страниц, требуемых для ovlp - 1
        inc     EAX         ;Число страниц, требуемых для ovlp
        mov     novlp,EAX   ;Сохраним его
        push    PAGEMAPGLOBAL;Вид блокирования
        push    EAX         ;Число блокируемых страниц
        push    ECX         ;Полный номер начальной блокируемой страницы
        VMMCall _LinPageLock;EAX=новый линейный адрес страницы с ovlp
        add     EAX,EBX     ;Новый линейный адрес ovlp
        mov     lpovlp,EAX  ;Сохраним его
        add     ESP,12      ;Восстановление стека
;Получим адрес буфера с данными настройки
        mov     EDI,dword ptr [ESI.lpvInBuffer]
;Засылаем управляющие слова по каналам
        mov     DX,303h     ;Регистр команд
        mov     AL,36h      ;Канал 0, режим 3
        out     DX,AL
        mov     AL,70h      ;Канал 1, режим 0
        out     DX,AL
        mov     AL,0B6h     ;Канал 2, режим 3
        out     DX,AL
;Программируем канал 0 - 1-ю половину таймера
        mov     AX,[EDI+0]  ;1-й параметр
        mov     DX,300h
        out     DX,AL       ;Младший байт частоты
        xchg    AL,AH
        out     DX,AL       ;Старший байт частоты
;Программируем канал 1 - 2-ю половину таймера
        mov     AX,[EDI+2]  ;2-й параметр
        mov     DX,301h
        out     DX,AL       ;Младший байт интервала
        xchg    AL,AH
        out     DX,AL       ;Старший байт интервала
;Программируем внутренний генератор
        mov     AX,[EDI+4]  ;3-й параметр
        mov     DX,302h
        out     DX,AL
        xchg    AH,AL
        out     DX,AL
;Сбрасываем счетчик
        mov     DX,308h
        out     DX,AL
;Установим флаг S2 разрешения счета
        mov     DX,30Bh
        in      AL,DX
;Размаскируем прерывания
        mov     EAX,IRQ_Handle
        VxDCall VPICD_Physically_Unmask
        clc
        ret
EndProc VMyD_Init
;-------------------------
BeginProc VMyD_Exit
;Разблокируем переменную Data приложения
        mov     EAX,lpdata  ;Адрес-псевдоним
        shr     EAX,12      ;Полный номер начальной страницы
        push    PAGEMAPGLOBAL
        push    ndata       ;Число страниц
```

```
        push   EAX          ;Полный номер начальной страницы
        VMMCall _LinPageUnLock
        add    ESP,12
;Разблокируем структуру ovlp приложения
        mov    EAX,lpovlp
        shr    EAX,12       ;Полный номер начальной страницы
        push   PAGEMAPGLOBAL
        push   novlp        ;Число страниц
        push   EAX          ;Полный номер начальной страницы
        VMMCall _LinPageUnLock
        add    ESP,12
        clc
        ret
EndProc VMyD_Exit
;-------------------------------
BeginProc VMyD_Int_13, High_Freq
        pushad
;Получим данное
        mov    DX,309h
        in     AL,DX
        mov    AH,AL
        dec    DX
        in     AL,DX
        mov    result,AX
;Сброс флагов готовности и разрешения счета
        mov    DX,30Ah
        out    DX,AL
;Выполним завершающие действия в PIC и выведем результаты
        mov    EAX,IRQ_Handle
        VxDCall VPICD_Phys_EOI
        VxDCall VPICD_Physically_Mask
        mov    AX,result
        mov    EDI,lpdata  ;Передадим данное
        mov    [EDI],AX    ;в буфер приложения
        mov    EAX,lpovlp  ;Адрес структуры OVERLAPPED
        mov    EBX,[EAX.O_Internal];Значение члена Internal
        VxDCall VWIN32_DIOCCompletionRoutine
        popad
        clc
        ret
EndProc VMyD_Int_13
VxD_CODE_ENDS
end VMyD_Real_Init
```

Этот вариант виртуального драйвера отличается от предыдущего двумя особенностями. Во-первых, в нем осуществляется блокирование в памяти физических страниц не только для данного Data, но и для структурной переменной ovlp, к которой также надо обеспечить доступ из обработчика аппаратных прерываний. Во-вторых, вызовом функции VWIN32_DIOCCompletionRoutine перед завершением обработчика прерываний драйвер оповещает приложение об окончании асинхронной операции. Этот вызов приводит к установке события и переводу потока с функцией Isr() в состояние выполнения.

Для обеспечения блокирования переменной ovlp адрес этой структуры, как уже отмечалось выше, пересылается в драйвер на этапе инициализации платы вместе с константами инициализации платы и адресом переменной Data. В сегменте данных драйвера резервируются двухсловные ячейки lpdata и lpovlp для адресов блокируемых переменных приложения, а также ndata и novlp для хранения числа блокируемых страниц.

В процедуре VMyD_Init драйвера теперь блокируются два участка физической памяти приложения: с переменной Data и со структурой ovlp.

Установка события, разблокирующая спящий поток с функцией Isr() приложения, осуществляется в конце процедуры VMyD_Init_13 обработчика аппаратных прерываний драйвера. После чтения результата измерений, посылки в контроллер прерываний команды EOI, маскирования нашего уровня прерываний и передачи в приложение (в переменную Data) результата измерений выполняется вызов функции VWIN32_DIOCCompletionRoutine, которую предоставляет системный виртуальный драйвер VWIN32.VXD. Этот вызов оповещает систему о завершении асинхронной операции в виртуальном драйвере и переводит в свободное состояние то событие, дескриптор которого содержится в элементе hEvent структуры OVERLAPPED (вспомним, что, создав событие, мы поместили его дескриптор hEvent в элемент ovlp.hEvent этой структуры). Функция VWIN32_DIOCCompletionRoutine требует, чтобы в регистре EBX содержалось то значение элемента Internal структуры OVERLAPPED, которое было передано в драйвер на этапе активизации механизма асинхронных операций. Поэтому вызов этой функции выглядит следующим образом:

```
mov    EAX,lpovl   ;Адрес структуры OVERLAPPED
mov    EBX,[EAX.O_Internal];Значение члена Internal
VxDCall VWIN32_DIOCCompletionRoutine
```

Для того чтобы обратиться к элементу Internal в структуре OVERLAPPED, надо знать символическое обозначение смещения этого элемента. Тот состав структуры OVERLAPPED, который был приведен выше и в котором фигурирует имя Internal, определен в файле WINBASE.H, который является заголовочным файлом для программ, написанных на языке Си. В программах на языке ассемблера его, естественно, использовать нельзя. Однако та же структура OVERLAPPED описана и в файле VWIN32.INC, входящем в состав пакета DDK и предназначенным для программ на языке ассемблера (только там она называется _OVERLAPPED). Первый элемент этой структуры носит имя O_Internal, которое мы и использовали в приведенном выше фрагменте.

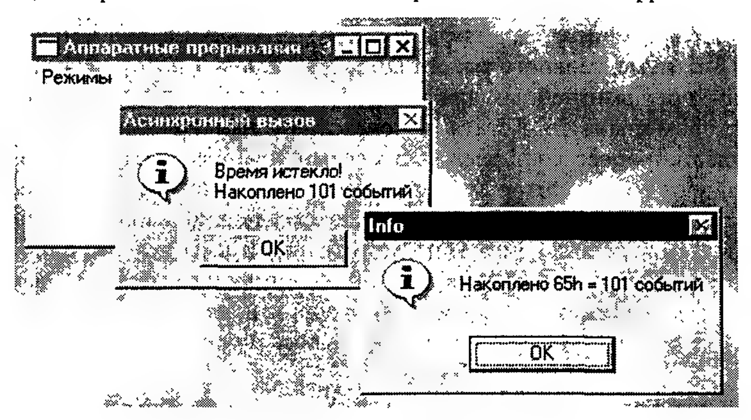


Рис. 87.2. Ход выполнения приложения 87-01

На рис. 87.2 показан результат выполнения программы. Видно, что после выбора пункта "Пуск" и истечения заданного интервала времени на экран было выведено окно сообщения из асинхронной функции Isr(). После этого для контроля был выбран пункт меню "Чтение данных", в котором вызывается прикладная функция ReadData(). Она, естественно, вывела из переменной Data то же число (101 событие).

Раздел восьмой

ПРИКЛАДНЫЕ ДРАЙВЕРЫ СИСТЕМ WINDOWS NT/2000

Статья 88. **Основы разработки прикладных драйверов Windows NT/2000**

Системы Windows NT/2000 (которые в дальнейшем мы для простоты будем называть Windows NT) заметно отличаются от более ранних вариантов операционных систем корпорации Microsoft как внутренними алгоритмами, так и терминологией. Системные программы, служащие для виртуализации аппаратуры и организации взаимодействия различных уровней операционной системы, в Windows NT носят название драйверов. Термин "виртуальный драйвер" остался за специфическими драйверами, используемыми главным образом для обслуживания сеансов DOS. Однако значение этих программ с точки зрения прикладного программиста осталось тем же: драйвер, работая в плоской модели памяти на нулевом уровне привилегий, имеет доступ ко всем ресурсам системы и компьютера и дает возможность решать задачи, недоступные обычным приложениям Windows. Одной из важнейших областей использования прикладных драйверов является разработка программных систем управления автоматизированными установками.

В отличие от виртуальных драйверов Windows 95/98, для разработки которых используется язык ассемблера, драйверы Windows NT, как правило, пишутся на языке Си. В состав инструментального пакета DDK NT включен набор необходимых макросов и функций, предназначенных для использования в Си-программах и обрабатываемых компилятором языка Си. Кстати, использование языка высокого уровня практически не упрощает процесс составления программ драйверов (во всяком случае, прикладных драйверов, для которых характерны относительно простые алгоритмы), однако существенно затрудняет их отладку. Действительно, наблюдая работу драйвера с помощью системного отладчика типа SoftICE, мы имеем дело не с операторами исходного текста драйвера, а с кодами машинных команд, которые не всегда легко сопоставить с предложениями языка Си. Таким образом, отладка Си-программы драйвера требует даже более глубокого владения языком ассемблера, чем разработка обычных прикладных программ на этом языке.

Все же включение в эту книгу, посвященную программированию на языке ассемблера, раздела, в котором рассматриваются исключительно Си-программы, выглядит несколько нелогично. Однако было бы странно, описывая методику составления прикладных драйверов для систем Windows, ограничиться лишь вариантами Windows 95/98. Поэтому авторы сочли необходимым рассказать также и о драйверах систем Windows на платформе NT.

В настоящей статье будет описан простейший драйвер, единственной функцией которого является передача в вызывающее его приложение Windows линейных адре-

сов своих процедур. Как уже отмечалось в предыдущем разделе, получение адресов процедур драйвера является практически обязательным условием его отладки, так как позволяет, установив в отладчике точку останова на адресе той или иной процедуры, войти в драйвер и продолжать его выполнение в пошаговом режиме, контролируя результат действия каждой команды. При этом необходимо иметь в виду, что в отличие от систем Windows 95/98, в которых системные программы, и в частности виртуальные драйверы, используют при работе в плоском адресном пространстве на нулевом уровне привилегий глобальные селекторы 0x28 для сегмента команд и 0x30 для сегмента данных, в системах NT тем же целям служат селекторы 0x08 и 0x23 (поскольку в этом разделе описываются программы, составленные на языке Си, 16-ричные числа мы будем обозначать по правилам этого языка, предваряя каждое 16-ричное число символами 0x). Таким образом, для того чтобы перейти в программу драйвера и продолжить ее выполнение в пошаговом режиме, следует, загрузив отладчик SofICE и работающее с драйвером приложение Windows, ввести в отладчике команду

G 8:*address*

где *address* – линейный 32-битовый адрес отлаживаемой функции, полученный с помощью того же драйвера. Разумеется, для того чтобы выполнить эту операцию, драйвер должен быть уже отлажен до такой степени, чтобы хотя бы выводить на экран правильные адреса своих функций.

Еще одно замечание относительно отладчика SoftICE. Для работы с системой Windows NT предусмотрен отдельный вариант отладчика. Он устанавливается и функционирует практически так же, как и для среды Windows 95/98, однако активизируется по-другому. После полной загрузки Windows NT следует запустить командный файл NTICE.BAT (удобно значок этого файла поместить на Рабочий стол); файл состоит из одной команды

NET START NTICE

Далее активизация отладчика выполняется запуском программы LOADER32.EXE, как и для систем Windows 95/98.

В процессе рассмотрения программы драйвера мы познакомимся с основами взаимодействия драйвера и системы Windows, а также опишем правила подготовки загрузочного модуля драйвера.

Состав и функционирование драйвера будут более понятны, если представлять, каким образом он вызывается, какие данные в него передаются и какой результат ожидает получить прикладная программа, работающая в паре с драйвером. Поэтому сначала мы рассмотрим текст этой прикладной программы (пример 88.1), которая является классическим 32-разрядным приложением Windows и в принципе не отличается от рассмотренных ранее примеров (см., например, статью 85). В последующих статьях этого раздела программа будет усложняться, главным образом за счет включения в ее меню дополнительных пунктов, а в текст программы – соответствующих этим пунктам прикладных функций. Структура и состав исходных файлов приложений будут оставаться неизменными. Приводимый текст приложения готовился в IDE Borland C++ 5.0; с равным успехом можно было воспользоваться и другой средой подготовки приложения Windows.

Пример 88.1. Приложение Windows, служащее для вызова драйвера, получения из него адресов его процедур и вывода этой информации на экран

Файл 88-01.H

```
//Определения констант
#define MI_ADDR 100
#define MI_EXIT 104
//Определение кода действия
#define IOCTL_ADDR (0x800<<2)|(0x22<<16)
//Прототипы функций
void Register(HINSTANCE);
void Create(HINSTANCE);
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
void GetAddr();
```

Файл 88-01.RC

```
#include "88-01.h"
Main MENU
{
  POPUP "Режимы"
  {
    MENUITEM "Адреса процедур драйвера",MI_ADDR
    MENUITEM SEPARATOR
    MENUITEM "Выход",MI_EXIT
  }
}
```

Файл 88-01.CPP

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <stdio.h>
#include "88-01.h"
char szClassName[]="MainWin";
char szTitle[]="Приложение 88-1";
HANDLE hDrv;//Дескриптор открытого драйвера
DWORD cbRet;//Счетчик байтов возвращаемых драйвером данных
//Главная функция приложения
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE,LPSTR,int){
  MSG msg;
  Register(hInstance);
  Create(hInstance);
  while(GetMessage(&msg,NULL,0,0))
        DispatchMessage(&msg);
  return 0;
  }
//Функция регистрации класса главного окна
void Register(HINSTANCE hInst){
  WNDCLASS wc;
  memset(&wc,0,sizeof(wc));
  wc.lpszClassName=szClassName;
  wc.hInstance=hInst;
  wc.lpfnWndProc=WndProc;
  wc.lpszMenuName="Main";
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  RegisterClass(&wc);
  }
```

```
//Функция создания и показа главного окна
void Create(HINSTANCE hInst){
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
                        10,10,300,150,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  }
//Оконная процедура главного окна
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg){
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
      return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
//Функция, вызываемая при создании главного окна
BOOL OnCreate(HWND,LPCREATESTRUCT){
//Откроем драйвер как файл и получим его дескриптор
  hDrv=CreateFile("\\\\.\\Win32Name",GENERIC_READ|GENERIC_WRITE,0,
                  NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
  return TRUE;
  }
//Функция, вызываемая при выборе пунктов меню
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id)  {
    case MI_ADDR://При выборе пункта "Адреса процедур драйвера"
      GetAddr();
      break;
    case MI_EXIT://При выборе пункта "Выход"
      DestroyWindow(hwnd);
    }
  }
//Функция завершения приложения
void OnDestroy(HWND){
  CloseHandle(hDrv);//Закроем драйвер при завершении приложения
  PostQuitMessage(0);
  }
//Функция получения адресов процедур драйвера
void GetAddr(){
  ULONG DrvAddr[4];//Приемный буфер (выходной для драйвера)
  char szText[80];
//Получим адреса процедур драйвера для отладки
  DeviceIoControl(hDrv,IOCTL_ADDR,NULL,0,DrvAddr,12,&cbRet,NULL);
  wsprintf(szText,
        "DriverEntry=%lX\nCtlCreate=%lX\nCtlClose=%lX\nCtlDispatch=%lX",
         DrvAddr[0],DrvAddr[1],DrvAddr[2],DrvAddr[3]);
  MessageBox(NULL,szText,"Адреса процедур драйвера",MB_ICONINFORMATION);
  }
```

В заголовочном файле 88-01.H содержатся, как это и положено, определения символических констант, используемых в программе, а также прототипы функций. Константы MI_ADDR и MI_EXIT служат идентификаторами пунктов меню приложения, а константа IOCTL_ADDR определяет значение задаваемого в приложении кода действия. Наш драйвер предназначен для выполнения лишь одного действия – возврата в приложение своих адресов; в последующих драйверах набор кодов действий будет расширяться. Некоторая вычурность определения константы IOCTL_ADDR связана с тем, что в Windows NT на значения кодов действий накладываются определенные условия, к которым мы еще вернемся при рассмотрении программы са-

мого драйвера. Пока только отметим, что числовое значение константы IOCTL_ADDR (представляющей собой длинное слово) составляет 0x00222000.

Файл ресурсов содержит сценарий меню, состоящего в данном случае всего из двух пунктов: вызова драйвера с целью получения адресов его процедур и завершения приложения.

Структура программы типична для несложных приложений Windows. Программа состоит из главной функции WinMain(), оконной процедуры WndProc и ряда вспомогательных функций. В оконной процедуре определяется состав сообщений Windows, обрабатываемых в данном приложении. Сообщение WM_CREATE посылается в главное окно в процессе его создания; на этом этапе (см. функцию OnCreate()) мы функцией Windows CreateFile() открываем драйвер. Сообщение WM_COMMAND, приводящее к вызову вспомогательной функции OnCommand(), приходит от пунктов меню приложения: при выборе пункта "Адреса процедур драйвера" вызывается функция GetAddr(), где происходит единственное в данной программе содержательное обращение к драйверу, а при выборе пункта "Выход" – завершение программы.

Функции CreateFile() передается целый ряд параметров. Первый параметр в данном случае не соответствует имени файла с программой драйвера. Использованное в качестве параметра (произвольное) имя Win32Name входит в "пространство имен Win32" и задается в тексте драйвера, который выполняет с ним определенные манипуляции. Имя файла драйвера вообще не фигурирует в тексте программ, хотя и заносится, как это будет показано позже, в реестр Windows.

Далее в параметрах функции CreateFile() указываются: тип доступа к устройству (в данном случае чтение-запись), флаг совместного использования (0 – запрет), атрибуты защиты (отсутствуют), действия системы в случае отсутствия открываемого файла (у нас – открыть драйвер при его наличии и ничего не делать в случае его отсутствия), атрибуты открываемого файла (у нас атрибуты отсутствуют) и, наконец, дескриптор дополнительного файла шаблона (который в данном случае тоже отсутствует).

Функция CreateFile() в случае своего успешного выполнения открывает драйвер и возвращает его дескриптор, сохраняемый в глобальной переменной hDrv типа HANDLE. Этот дескриптор даст нам возможность обращаться к драйверу с помощью функции IOCTL-интерфейса DeviceIoControl().

Как уже отмечалось, единственное содержательное обращение к драйверу осуществляется при выборе пункта меню "Адреса процедур драйвера" с идентификатором MI_ADDR. В этом случае вызывается подпрограмма GetAddr(), содержащая вызов функции DeviceIoControl() с указанием единственного в нашей программе кода действия с символическим обозначением IOCTL_ADDR. Как уже описывалось в статье 83, функции DeviceIoControl() вслед за дескриптором драйвера и кодом действия передаются адреса и размеры двух буферов обмена данными: входного (с точки зрения драйвера), через который приложение может передать в драйвер необходимую информацию, и выходного, в который драйвер помещает данные, передаваемые в приложение. В данном случае приложение ничего не передает в драйвер, поэтому вместо адреса буфера и его размера указаны нулевые значения, а в качестве выходного для драйвера и входного для программы буфера указан массив DrvAddr из четырех длинных чисел, в котором приложение после возврата из функции DeviceIoControl() ожидает найти линейные адреса четырех процедур драйвера.

Полученные из драйвера адреса преобразуются в символьную форму и выводятся в окно сообщения. Как уже отмечалось в предыдущих статьях, драйвер может поместить в буфер обмена данные любых типов и в любом порядке. Чтобы правильно извлечь данные, приложение должно знать, как они размещены в буфере. В данном случае драйвер записывает в буфер 4 длинных адреса и для их приема в программе проще всего объявить массив длинных слов DrvAddr из четырех элементов. Порядок выборки данных из буфера приложением, разумеется, должен соответствовать порядку их помещения в буфер драйвером.

Возможный вывод программы приведен на рис. 88.1. Обратите внимание на странное совпадение адресов двух различных функций драйвера – CtlCreate() и CtlClose(). Как две разные функции могут иметь один и тот же адрес? При рассмотрении программы драйвера мы вернемся к вопросу об этом совпадении.

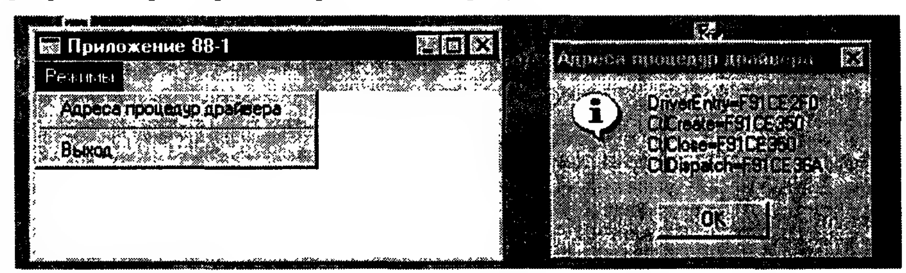


Рис. 88.1. Вывод приложения 88-01 с окном сообщения и открытым меню

При завершении приложения выполняется закрытие драйвера с помощью функции Windows CloseHandle() с указанием в качестве параметра дескриптора открытого ранее драйвера. Вызов этой функции не является обязательным, так как в процессе завершения приложения Windows закрывает все открытые дескрипторы.

Перейдем теперь к программе драйвера. Помимо собственно текста программы драйвера мы рассмотрим основные алгоритмы взаимодействия системы Windows, драйвера и приложения, а также некоторые системные структуры данных, используемые драйвером в процессе его функционирования. В тексте программы драйвера приходится постоянно ссылаться на те или иные элементы этих структур, поэтому программист должен иметь отчетливое представление об их составе и назначении пример 88.1, продолжение).

Пример 88.1 (продолжение). Простой драйвер, посылающий в приложение адреса своих характерных процедур

```
#include "ntddk.h"
#define NT_DEVICE_NAME     L"\\Device\\DeviceName"//Имя объекта устройства
#define WIN32_DEVICE_NAME  L"\\??\\Win32Name"//Имя в пространстве имен Win32
//Определим код действия при вызове драйвера
#define IOCTL_ADDR CTL_CODE \
   (FILE_DEVICE_UNKNOWN,0x800,METHOD_BUFFERED,FILE_ANY_ACCESS)
//Прототипы основных функций драйвера
NTSTATUS CtlCreate(IN PDEVICE_OBJECT,IN PIRP);
NTSTATUS CtlClose(IN PDEVICE_OBJECT,IN PIRP);
NTSTATUS CtlDispatch(IN PDEVICE_OBJECT,IN PIRP);
//Функция инициализации драйвера
NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
                     IN PUNICODE_STRING pRegistryPath){
  PDEVICE_OBJECT pDeviceObject;//Указатель на объект устройства
```

```
  UNICODE_STRING uniNtName;//Структура с NT-именем устройства
  UNICODE_STRING uniWin32Name;//Структура с Win32-именем устройства
//Преобразуем оба имени в структурные переменные со счетчиками
  RtlInitUnicodeString(&uniNtName,NT_DEVICE_NAME);
  RtlInitUnicodeString(&uniWin32Name,DOS_DEVICE_NAME);
//Создадим символическую связь NT- и Win32-имен
  IoCreateSymbolicLink(&uniWin32Name,&uniNtName);
//Создадим объект устройства
  IoCreateDevice(pDriverObject,//Указатель на объект драйвера
                 0,//Расширение отсутствует
                 &uniNtName,//Имя устройства
                 FILE_DEVICE_UNKNOWN,//Тип устройства
                 0,//Специальные характеристики устройства
                 FALSE,//Однопоточное устройство
                 &pDeviceObject);//Указатель на создаваемый объект
//Заполним в структуре объекта драйвера поля с адресами основных функций
  pDriverObject->MajorFunction[IRP_MJ_CREATE]=CtlCreate;
  pDriverObject->MajorFunction[IRP_MJ_CLOSE]=CtlClose;
  pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]=CtlDispatch;
  return STATUS_SUCCESS;//Успешное завершение функции
  }
//Функция, вызываемая при открытии драйвера приложением
NTSTATUS CtlCreate(IN PDEVICE_OBJECT pDeviceObject,IN PIRP pIrp){
//В данном драйвере не выполняем никаких действий
  pIrp->IoStatus.Status=STATUS_SUCCESS;//Код успешного завершения
  pIrp->IoStatus.Information=0;//Дополнительная информация
  IoCompleteRequest(pIrp,IO_NO_INCREMENT);//Завершение данного запроса
  return STATUS_SUCCESS;//Успешное завершение функции
  }
//Функция, вызываемая при закрытии драйвера приложением
NTSTATUS CtlClose(IN PDEVICE_OBJECT pDeviceObject,IN PIRP pIrp){
//В данном драйвере не выполняем никаких действий
  pIrp->IoStatus.Status=STATUS_SUCCESS;//Код успешного завершения
  pIrp->IoStatus.Information=0;//Дополнительная информация
  IoCompleteRequest( pIrp,IO_NO_INCREMENT);//Завершение данного запроса
  return STATUS_SUCCESS;//Успешное завершение функции
  }
//Функция диспетчеризации
NTSTATUS CtlDispatch(IN PDEVICE_OBJECT DeviceObject,IN PIRP pIrp){
  PIO_STACK_LOCATION pIrpStack;//Указатель на стековую область
  PULONG pIOBuffer;//Системный буфер обмена данными с приложением
  pIrpStack=IoGetCurrentIrpStackLocation(pIrp);//Получим адрес стековой области
  pIOBuffer=pIrp->AssociatedIrp.SystemBuffer;//Получим адрес буфера обмена
  switch(pIrpStack->Parameters.DeviceIoControl.IoControlCode){
    case IOCTL_ADDR://Код действия
      *pIOBuffer++=(ULONG)DriverEntry;//В буфер обмена адрес функции DriverEntry
      *pIOBuffer++=(ULONG)CtlCreate;//В буфер обмена адрес функции CtlCreate
      *pIOBuffer++=(ULONG)CtlClose;//В буфер обмена адрес функции CtlClose
      *pIOBuffer=(ULONG)CtlDispatch;//В буфер обмена адрес функции CtlDispatch
      pIrp->IoStatus.Information=16;//Число передаваемых байтов
      break;
    }
  pIrp->IoStatus.Status=STATUS_SUCCESS;//Код успешного завершения
  IoCompleteRequest(pIrp,IO_NO_INCREMENT );//Завершение данного запроса
  return STATUS_SUCCESS;//Успешное завершение функции
  }
```

Текст драйвера начинается с оператора препроцессора #include, с помощью которого к программе подсоединяется файл NTDDK.H, содержащийся в пакете DDK NT. Этот файл включает значительную часть определений констант, типов переменных,

прототипов функций и макросов, используемых в исходных текстах программ драйверов. Некоторая доля этих определений входит в другие заголовочные файлы, на которые имеются ссылки в файле NTDDK.H.

Два следующих предложения программы служат для задания символических имен (в данном случае NT_DEVICE_NAME и WIN32_DEVICE_NAME) текстовым строкам с именами объекта устройства, который будет создан нашим драйвером. Вопрос об объекте устройства и его именах будет подробнее рассмотрен ниже.

Предложения вида

```
#define IOCTL_ADDR CTL_CODE \
  (FILE_DEVICE_UNKNOWN,0x800,METHOD_BUFFERED,FILE_ANY_ACCESS)
```

позволяют определить коды действий, выполняемых драйвером по запросам приложения. Как уже отмечалось в статье 83, обращение приложения к драйверу, независимо от цели этого обращения, осуществляется с помощью единой функции DeviceIoControl(). Для того чтобы приложение могло запросить у драйвера выполнение конкретного действия (из числа предусмотренных в драйвере), в качестве одного из параметров этой функции выступает код действия (в данном случае IOCTL_ADDR). Процедура драйвера, вызываемая функцией Windows DeviceIoControl(), должна проанализировать поступивший в драйвер код действия и передать управление на соответствующий фрагмент драйвера.

Коды действия, называемые в документации DDK NT управляющими кодами ввода-вывода (I/O control codes), строятся по определенным правилам. Каждый код представляет собой слово длиной 32 бита, в отдельных полях которого размещаются компоненты кода (рис. 88.2).

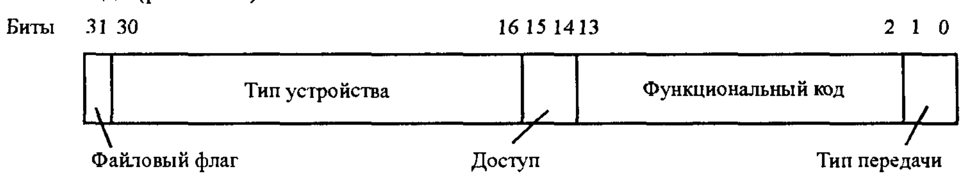


Рис. 88.2. Поля кода действия

Файловый флаг устанавливается в случаях, когда пользователь создает новые коды действия для файловых устройств. Все наши устройства нефайловые, и этот бит должен быть сброшен.

В поле "Тип устройства" помещается предопределенная константа, характеризующая устройство (FILE_DEVICE_CD_ROM, FILE_DEVICE_MOUSE и др.). В нашем случае можно использовать константу FILE_DEVICE_UNKNOWN, равную 0x22.

Поле доступа определяет запрашиваемые пользователем права доступа к устройству (чтение, запись, чтение и запись). Мы будем использовать константу FILE_ANY _ACCESS, равную нулю.

Функциональный код может принимать произвольное значение в диапазоне 0x800...0xFFF (значения 0x000...0x7FF зарезервированы для кодов Microsoft). В рассматриваемом примере используется единственный код действия и для него выбран функциональный код, равный 0x800. В последующих примерах драйверов кодов действий будет больше и им будут присваиваться функциональные коды 0x801, 0x802 и т. д.

Тип передачи определяет способ связи приложения с драйвером. Для драйверов физических устройств, выполняющих пересылки незначительных объемов данных без

использования канала прямого доступа, в качестве типа передачи обычно используется константа METHOD_BUFFERED, равная нулю. Такой выбор константы определенным образом задает местоположение системного буфера, через который пересылаются данные. В дальнейшем этот вопрос будет рассмотрен подробнее.

Код действия можно сформировать "вручную", как это сделано в нашем приложении Windows:

```
#define IOCTL_ADDR (0x800<<2)| (0x22<<16)
```

Легко сообразить, что в этом случае предполагаются константы FILE_DEVICE_UNKNOWN=0x22, METHOD_BUFFERED=0 и FILE_ANY_ACCESS=0 при значении функционального кода 0x800. В программе драйвера для формирования кода действия использован макрос CTL_CODE, который определен в файле NTDDK.H. Этот макрос позволяет обойтись без детального знания формата кода действия и значений конкретных констант.

Вслед за определением кода действия в тексте драйвера приведены прототипы используемых в нем функций. Этим функциям можно дать произвольные имена, однако их, как говорят, сигнатура, т. е. состав параметров вместе с типом возвращаемого значения, жестко заданы системой. Ключевое слово IN, с которого начинается описание каждого параметра, говорит о том, что этот параметр является для функции входным, т. е. передается в функцию при ее вызове. В других случаях может использоваться ключевое слово OUT, а также и комбинация IN OUT, если через данный параметр осуществляется как передача данного в функцию, так и возврат результата ее работы. По правилам языка Си, функция может изменять значения передаваемых в нее через параметры данных, если параметром является не само данное, а его адрес (указатель).

Программная часть драйвера начинается с обязательной функции с именем DriverEntry(), которая автоматически вызывается системой на этапе загрузки драйвера и должна содержать все действия по его инициализации. В первых строках функции определяются используемые в ней данные – указатель на объект устройства типа PDEVICE_OBJECT и две символьные строки типа UNICODE_STRING с именами устройства. В качестве первого параметра функция получает указатель еще на один объект, именно на объект драйвера типа PDRIVER_OBJECT. О каких объектах идет речь и почему объект устройства имеет два имени?

Windows NT является объектно-ориентированной системой. Каждый компонент системы представляет собой объект, включающий в себя необходимые для его функционирования структуры данных и наборы функций. Некоторые из этих функций служат для внутреннего использования данным объектом, другие же являются экспортируемыми, т. е. доступными другим объектам. Системные компоненты общаются друг с другом не напрямую, а исключительно с помощью экспортируемых объектами функций.

Типы объектов Windows, т. е. состав входящих в них структур данных и функций, известны заранее, однако сами объекты (или, как говорят, экземпляры объектов) создаются динамически по мере возникновения в них необходимости.

При загрузке драйвера система создает объект драйвера (driver object), олицетворяющий для системы образ драйвера в памяти. С другой стороны, объект драйвера представляет собой структуру, содержащую необходимые для функционирования драйвера данные и адреса (указатели) функций.

В процессе инициализации драйвера (процедуру инициализации пишет программист-разработчик драйвера) создаются один или несколько объектов устройств (device object), олицетворяющих те устройства, с которыми будет работать данный драйвер. Объекты устройств существуют все время, пока драйвер находится в памяти; если мы не предусматриваем специальных средств динамической выгрузки драйвера, то объекты устройств будут уничтожены лишь при завершении работы системы Windows. Объект устройства (по крайней мере один) необходим для правильного функционирования драйвера и создается даже в том случае, если, как это имеет место в нашем примере, драйвер не имеет отношения к каким-либо реальным физическим устройствам.

Системные программы взаимодействуют с объектом устройства, созданным драйвером, посредством указателя на него. Однако для прикладной программы объект устройства представляется одним из файловых объектов (вспомним, что первым обращением к драйверу в приложении был вызов функции CreateFile()) и обращение к нему осуществляется по имени. Вот это-то имя, в нашем случае Win32Name, и следует определить в программе драйвера.

Дело усугубляется тем, что объект устройства должен иметь два имени, одно – в пространстве имен NT, другое – в пространстве имен Win32. Оба эти имени должны, во-первых, быть определены с помощью кодировки Unicode, в которой под каждый символ выделяется не 1, а 2 байта, и, во-вторых, представлять собой не просто символьные строки, а специальные структуры типа UNICODE_STRING, в которые входят помимо самих строк еще и их длины ("структуры со счетчиками"). Кодировка Unicode задается с помощью символа L, помещаемого перед символьной строкой в кавычках, а преобразование строк символов в структуры типа UNICODE_STRING осуществляется вызовами функции RtlInitUnicodeString(), которые можно найти далее по тексту программы драйвера.

Имена объектов устройств составляются по определенным правилам. NT-имя предваряется префиксом \Device\, а Win32-имя – префиксом \??\ (или \DosDevice\). При указании имен в Си-программе знак обратной косой черты удваивается.

Для того чтобы указанное в программе драйвера имя можно было использовать в приложении для открытия устройства, следует создать символическую связь между обоими заданными именами устройства. Эта связь создается функцией IoCreateSymbolicLink(), которой в качестве параметров передаются оба имени.

Следующая обязательная операция – создание объекта устройства – осуществляется вызовом функции IoCreateDevice(), принимающей ряд параметров. Первый параметр, указатель на объект драйвера, поступает в функцию DriverEntry() при ее вызове из Windows (см. заголовок функции DriverEntry). Второй параметр определяет размер так называемого расширения устройства – области, служащей для передачи данных между функциями драйвера. В рассматриваемом драйвере расширение устройства не используется и на месте этого параметра указан 0. В качестве третьего параметра указывается созданное нами ранее NT-имя устройства. Наконец, последний параметр этой функции является выходным – через него функция возвращает указатель (типа DEVICE_OBJECT) на созданный объект устройства.

Последнее, что надо сделать на этапе инициализации драйвера, – это занести в объект драйвера адреса основных функций, включенных программистом в текст драйвера. Под основными функциями мы будем понимать те фрагменты драйвера, которые вызываются системой автоматически в ответ на определенные действия, выполняемые

приложением или устройством. В наших примерах драйверов таких действий будет три: получение дескриптора драйвера функцией CreateFile(), запрос к драйверу на выполнение требуемого действия функцией DeviceIoControl() и закрытие драйвера функцией CloseHandle(). В более сложных драйверах основных функций может быть больше (вплоть до приблизительно трех десятков). Для хранения адресов основных функций в объекте драйвера предусмотрен массив (с именем MajorFunction) указателей на функции типа PDRIVER_DISPATCH. В файле NTDDK.H определены символические смещения элементов этого массива. Так, в первом элементе массива (смещение IRP_MJ_CREATE=0) должен размещаться указатель на функцию, которая вызывается автоматически при выполнении в приложении функции CreateFile(). В элементе со смещением IRP_MJ_CLOSE=2 размещается указатель на функцию, вызываемую при закрытии устройства (функцией CloseHandle()). Наконец, в элементе со смещением IRP_MJ_DEVICE_CONTROL=0x0E должен находиться адрес функции диспетчеризации, которой система передает управление в ответ на вызов в выполняемом приложении Windows функции DeviceIoControl() с указанием кода требуемого действия. Назначение функции диспетчеризации – анализ кодов действий, направляемых в драйвер приложением, и осуществление переходов на соответствующие фрагменты драйвера. В рассматриваемом примере три упомянутые функции имеют (произвольные) имена CtlCreate, CtlClose и CtlDispatch; структура нашего драйвера с указанием его функций точек входа приведена на рис. 88.3.

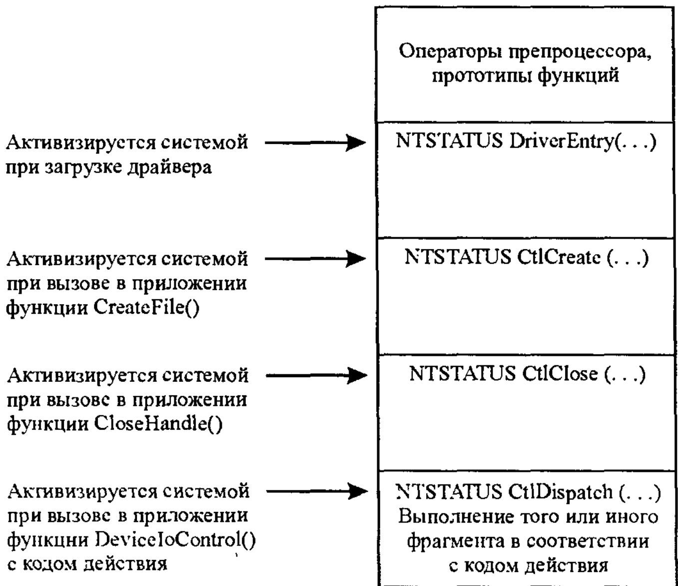


Рис. 88.3. Структура простейшего драйвера

Массив MajorFunction является одним из элементов структурной переменной. Если бы эта структура была объявлена в программе с указанием ее имени (пусть это имя будет DriverObject), то для обращения к элементу структуры с индексом 0 следовало бы использовать конструкцию с символом точки:

```
DriverObject.MajorFunction[0]=CtlCreate;
```

Однако у нас имеется не имя структурной переменной, а ее адрес pDriverObject, полученный в качестве первого параметра при активизации функции DriverEntry. В этом случае для обращения к элементу структуры следует вместо точки использовать обозначение ->:

```
pDriverObject->MajorFunction[IRP_MJ_CREATE]=CtlCreate;
```

Разумеется, вместо численного значения индекса массива надежнее воспользоваться символическим.

Функция DriverEntry(), как, впрочем, и все остальные функции, входящие в состав драйвера, завершается оператором return с указанием кода успешного завершения STATUS_SUCCESS (равного нулю).

Как видно из прототипов функций CtlCreate(), CtlClose() и CtlDispatch(), все они принимают (из системы Windows) в качестве первого параметра указатель на объект драйвера, а в качестве второго – указатель на структуру типа IRP. Эта структура, так называемый пакет запроса ввода-вывода (in/out request packet, IRP), играет чрезвычайно важную роль в функционировании драйвера наряду с уже упоминавшимися объектами драйвера и устройства. Рассмотрим более детально создание и взаимодействие всех этих структур (рис. 88.4).

Как уже упоминалось выше, объект драйвера, олицетворяющий собой образ выполнимой программы драйвера в памяти, создается при загрузке драйвера на этапе запуска системы Windows. В этом объекте еще не заполнен массив MajorFunction, а также DeviceObject – указатель на объект устройства, поскольку сам объект устройства пока еще не существует.

Загрузив драйвер, Windows активизирует его функцию инициализации DriverEntry(). Эта функция должна содержать вызов IoCreateDevice(), создающий объект устройства. В объекте устройства есть ссылка на объект драйвера, которому это устройство принадлежит, и, кроме того, адрес так называемого расширения устройства (device extension), поля произвольного размера, служащего для обеспечения передачи данных между запросами ввода-вывода. В настоящем примере драйвера расширение устройства не используется (и соответственно, не создается). В дальнейших примерах будет показано, как задать формат и размеры расширения устройства, а также каким образом использовать его возможности. Функция IoCreateDevice(), создав объект устройства, заносит его адрес в объект драйвера. Таким образом, обе эти структуры оказываются взаимосвязаны.

Рассмотренные выше объекты существуют независимо от запуска и функционирования прикладной программы, работающей с драйвером. Уничтожены они будут только при закрытии всей системы или при динамической выгрузке драйвера из памяти, если такая возможность в драйвере предусмотрена.

Пакет запроса ввода-вывода создается заново при каждом обращении приложения к драйверу, т. е. при каждом вызове функции DeviceIoControl(). В терминологии драйверов Windows NT этот вызов носит название запроса ввода-вывода (I/O request). Выполнение функции IoCompleteRequest(), которой завершается любая активизируемая из приложения функция драйвера, приводит к уничтожению этого пакета, который, таким образом, существует лишь в течение времени выполнения активизированной функции драйвера. Обычно приложение за время своей жизни обращается к драйверу неоднократно; следующий запрос ввода-вывода снова создаст пакет IRP, который, разумеется, ничего не будет знать о предыдущем. Для того чтобы можно было передать

данные, полученные в одном запросе ввода-вывода (например, данные, прочитанные из устройства), в другой запрос (например, с целью записи их в устройство), эти данные следует сохранить в расширении устройства.

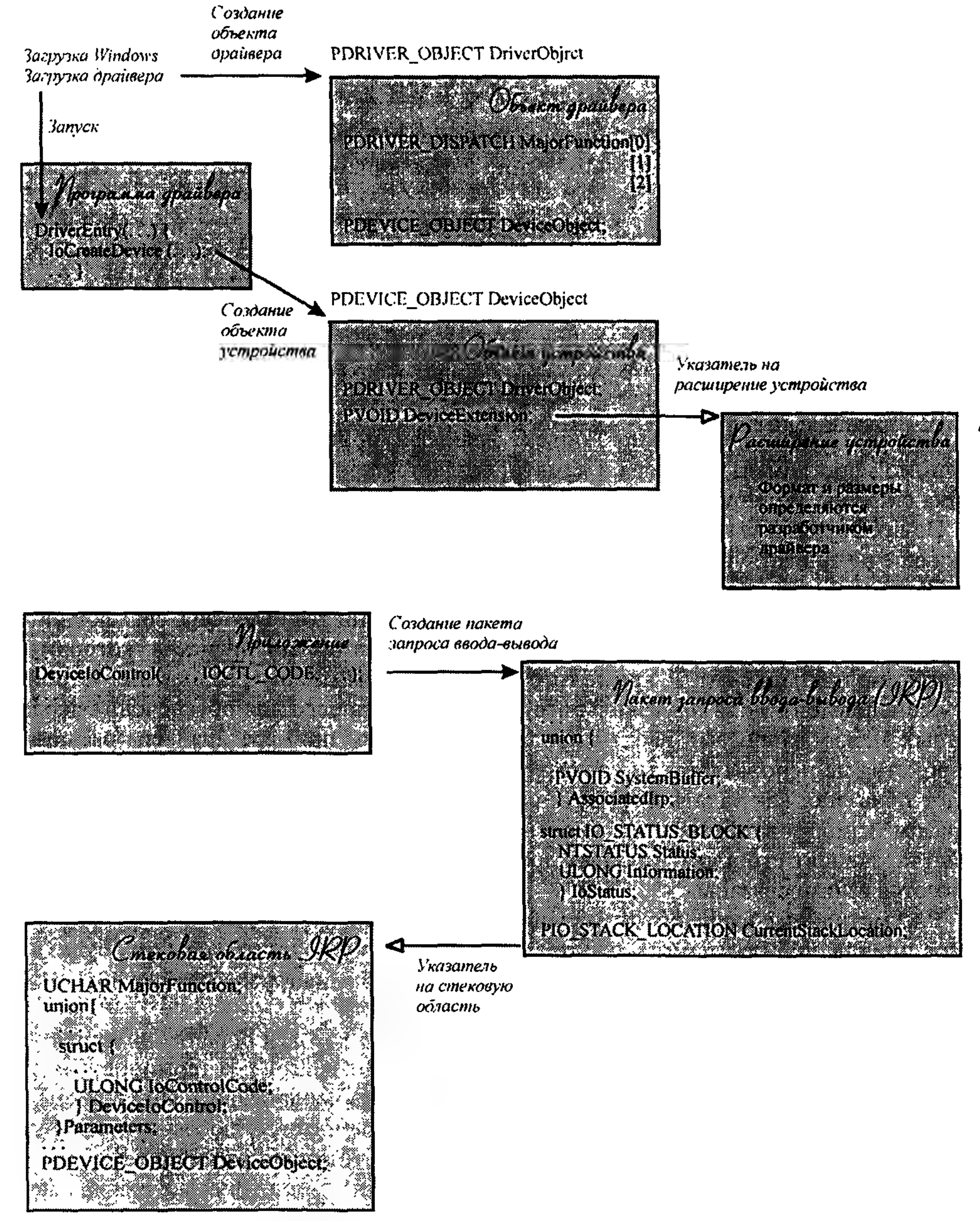


Рис. 88.4. Структуры данных, обеспечивающие работу драйвера Windows NT

Пакет ввода-вывода IRP состоит из двух частей: фиксированной части и так называемой стековой области ввода-вывода (I/O stack location). Как видно из рис. 88.4, ссылка на стековую область ввода-вывода содержится в переменной CurrentStack Location, входящей в фиксированную часть IRP. Адрес же фиксированной части передается в качестве второго параметра в любую основную функцию драйвера при ее активизации. С другой стороны, адрес стековой области ввода-вывода можно получить с помощью специально предусмотренной функции IoGetCurrentIrpStackLocation().

Итак, в нашем драйвере имеются три основные функции: CtlCreate(), CtlClose() и CtlDispatch(). Это минимальный набор основных функций, при котором драйвер будет правильно функционировать. Активизация каждой из этих функций приводит к выделению системой (конкретно – диспетчером ввода-вывода I/O Manager) пакета запроса ввода-вывода со стековой областью, а завершение функции – к его возврату в систему. Для освобождения пакета запроса ввода-вывода необходимо заполнить в нем структуру блока состояния IO_STATUS_BLOCK и сообщить диспетчеру ввода-вывода вызовом функции IoCompleteRequest() о том, что мы завершили обработку этого пакета. Как видно из рис. 88.4, в блок состояния входят две переменных: Status – для кода завершения и Information – для возврата в приложение некоторой числовой информации. В переменную Status естественно поместить код STATUS_SUCCESS, а переменная Information должна содержать число пересылаемых в приложение байтов. Функции CtlCreate() и CtlClose() ничего не пересылают в приложение, и значение этой переменной приравнивается нулю; функция CtlDispatch должна вернуть в приложение четыре 4-байтовых адреса, и переменной Information задается значение 16.

Функция IoCompleteRequest() требует указания двух параметров: указателя на текущий пакет запроса ввода-вывода и величины, на которую следует повысить приоритет вызывающей драйвер программы. В нашем случае запросы ввода-вывода обрабатываются очень быстро, за это время приоритет вызывающей программы снизиться не успевает, и нет необходимости его повышать. Поэтому в качестве второго параметра передается константа IO_NO_INCREMENT.

Функции CtlCreate() и CtlClose() в нашем примере не выполняют никакой содержательный работы, и их тексты в результате оказались полностью совпадающими. Транслятор с языка Си обратил на это внимание и удалил из текста программы один из повторяющихся участков, назначив именам CtlCreate() и CtlClose() один и тот же адрес. Поэтому в выводе программы (см. рис. 88.1) у обеих функций оказались совпадающие адреса. Вообще следует отметить, что трансляторы с языков высокого уровня, выполняя оптимизацию программы, иной раз существенно изменяют ее структуру. В результате отладка такой программы на уровне машинных кодов, которую приходится проводить, если драйвер работает слишком непредсказуемо, сопряжена с определенными трудностями.

Перейдем к рассмотрению содержательной (с точки зрения прикладного программиста) части драйвера – функции диспетчеризации CtlDispatch().

Как видно из рис. 88.4, системный буфер, служащий для обмена информацией между драйвером и приложением, расположен в пакете запроса ввода-вывода IRP (переменная SystemBuffer). Таким образом, для передачи в приложение адресов функций драйвера необходимо получить доступ к IRP, а через IRP – к SystemBuffer. С этой целью в функции CtlDispatch() объявляется переменная pIrpStack типа указателя на стековую область ввода-вывода PIO_STACK_LOCATION и, кроме того, переменная

pIOBuffer, в которую будет помешен адрес системного буфера обмена. В структуре пакета запроса ввода-вывода этот адрес имеет тип PVOID – указатель на переменную произвольного типа. Действительно, тип передаваемых в приложение (или из приложения) данных может быть каким угодно: он определяется конкретными задачами данного запроса ввода-вывода. В нашем примере мы передаем через буфер обмена адреса функций, представляющих собой двойные слова, поэтому и для переменной pIOBuffer выбрали тип PULONG – указатель на длинные слова без знака.

С помощью функции IoGetCurrentStackLocation() в переменную pIrpStack помещается адрес стековой области ввода-вывода, а затем в переменную pIOBuffer заносится адрес системного буфера из структуры IRP. Из рис. 88.4 видно, что системный буфер входит в объединение (union) с именем AssociatedIrp, поэтому для доступа к переменной SystemBuffer использована конструкция pIrp->AssociatedIrp.SystemBuffer. Объединение можно рассматривать как эквивалент структурной переменной с тем отличием, что все члены объединения размещаются (альтернативно) в одной и той же области памяти. В синтаксическом плане обращения к объединению и к структуре выполняются одинаково.

Конструкция switch-case анализирует содержимое ячейки IoControlCode, входящей в стековую область IRP (см. рис. 88.4) и в зависимости от значения кода действия, содержащегося в этой ячейке, передает управление на тот или иной фрагмент программы драйвера. В рассматриваемом примере предусмотрен единственный код действия IOCTL_ADDR, и его анализ не имеет смысла (и соответственно, операторы switch и case не нужны), однако мы предпочли включить в программу эту стандартную конструкцию, которая в последующих примерах будет использоваться уже вполне оправданно.

Засылка в буфер обмена четырех адресов функций драйвера осуществляется через указатель на этот буфер путем операции "снятия ссылки" (по правила языка Си, если pIOBuffer – адрес некоторой ячейки, то *pIOBuffer обозначает саму ячейку). Инкремент адреса после каждой операции пересылки перемещает указатель на следующее слово буфера обмена. Естественно, последняя пересылка не требует модификации адреса. Каждый пересылаемый адрес предварительно преобразуется в тип ULONG – длинное слово без знака.

В последнем предложении блока case в переменную IoStatus.Information пакета IRP заносится число пересылаемых байтов (16). Это данное поступит в Windows-приложение в переменную cbRet, указанную в качестве одного из параметров вызова функции DeviceIoControl(). При необходимости в приложении можно проанализировать значение этой переменной и учесть его в процессе дальнейшего выполнения программы. Мы этого не делаем.

В заключение одно замечание, можно сказать, лингвистического характера. В наших примерах имена переменных, обозначающих адреса каких-либо объектов, начинаются с символа p (от pointer, указатель). Этот метод обозначения переменных, при котором в имени любой переменной содержится некоторая информация о ее типе ("венгерская нотация") был предложен в свое время программистами Microsoft и широко используется как в технической документации, так и в прикладных программах. Однако при составлении файла NTDDK.H и других включаемых файлов пакета DDK NT этот метод использован не был. Так, например, имя DeviceObject обозначает не сам объект устройства, а указатель на него. С другой стороны, в обозначениях типов переменных венгерская нотация выдерживается:

```
PDEVICE_OBJECT DeviceObject;  /*Переменная типа
                               "указатель на структуру типа DEVICE_OBJECT"*/
PIO_STACK_LOCATION CurrentStackLocation;/*Переменная типа
                        "указатель на структуру типа IO_STACK_LOCATION"*/
UCHAR MajorFunction; /*Переменная типа UCHAR (unsigned char,
                               символьная без знака)*/
PUCHAR Buffer;//Переменная типа "указатель на переменную типа UCHAR"
```

На приведенном выше рис. 88.4 использованы фрагменты из документации, поэтому обозначения некоторых переменных оказались не такими, как в тексте программы драйвера.

Для работы с прикладными драйверами необходимо создать соответствующую операционную среду. Прежде всего в системе должен быть установлен пакет DDK NT, который даст возможность пользоваться документацией (которая, впрочем, входит и в пакет Visual Studio или, точнее, в справочник MSDN этого пакета) и, главное, рядом инструментальных программ: ML.EXE, BUILD.EXE и др. В процессе установки произойдет настройка переменных окружения; полезно проконтролировать результат установки DDK и убедиться, что в окружение включены следующие переменные (в предположении, что пакет DDK установлен в каталоге DDK диска I:):

```
BASEDIR=I:\DDK
DDKDRIVE=I:
DDKBUILDENV=FREE
NTMAKEENV=I:\DDK\INC
PATH=...;I:\DDK\BIN
```

Установочная программа DDK NT потребует, чтобы на компьютере предварительно был установлен пакет SDK WIN32. При его отсутствии можно воспользоваться пакетом Visual Studio; в этом случае следует включить в окружение переменную INCLUDE с описанием пути к каталогу INCLUDE пакета Visual Studio, а также убедиться в том, что переменная PATH дополнена всеми необходимыми путями к выполнимым файлам этого пакета.

Будем считать, что файл с исходным текстом программы драйвера называется APPDRV.C и находится в каталоге F:\DRIVERS. Для подготовки загрузочного модуля драйвера и установки его в системе необходимо подготовить три инструментальных файла (в качестве образцов можно воспользоваться файлами какого-либо из примеров, включенных в состав DDK NT, например из каталога DDK\SRC\GENERAL\SIMPLE \SYS): MAKEFILE, SOURCES и APPDRV.INI. Эти файлы должны находиться в том же каталоге, где расположен файл с исходным текстом драйвера. Инструментальные файлы имеют следующий состав:

Файл MAKEFILE

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

Файл SOURCES

```
TARGETNAME=appdrv
TARGETPATH=f:\drivers
TARGETTYPE=DRIVER
SOURCES=appdrv.c
```

Файл APPDRV.INI

```
\registry\machine\system\currentcontrolset\services\AppDrv
   Type = REG_DWORD 0x00000001
   Start = REG_DWORD 0x00000002
   Group = Extended base
   ErrorControl = REG_DWORD 0x00000001
```

Трансляция программы драйвера выполняется в сеансе DOS с помощью команды

```
BUILD -ce
```

При первом выполнении команды BUILD в рабочем каталоге (у нас это каталог F:\DRIVERS) будет создана разветвленная система подкаталогов: OBJ\I386 и I386\FREE; выполнимый модуль драйвера (в нашем случае APPDRV.SYS) будет помещен в подкаталог I386\FREE. Кроме этих подкаталогов процедуре трансляции понадобится подкаталог с именем NT в корневом каталоге рабочего диска (в нашем случае диска F:), куда будет помещен вспомогательный файл данных BUILD.DAT.

При наличии в исходном тексте драйвера серьезных ошибок в рабочем каталоге будет создан файл BUILD.ERR с перечнем ошибок; при наличии предупреждений – файл BUILD.WRN. При повторной трансляции (после устранения ошибок) эти файлы уничтожаются автоматически.

Получив выполнимый модуль драйвера (APPDRV.SYS), следует выполнить две операции: скопировать его в каталог \NT\SYSTEM32\DRIVERS и выполнить команду

```
REGINI APPDRV.INI
```

Эта команда занесет информацию о драйвере в реестр Windows. Для того чтобы новый драйвер был включен в систему, необходимо выполнить перезагрузку Windows.

Информация о драйвере, записанная в реестре, останется там навсегда – до удаления ее вручную с помощью программ RegEdit или RegEdt32. Поэтому удобно все отлаживаемые варианты драйвера называть одинаково; в этом случае после трансляции нового варианта достаточно скопировать драйвер в системный каталог Windows и перезагрузить систему, вызывать же каждый раз программу RegIni не требуется.

Статья 89. **Драйвер для работы с физической памятью**

Как уже отмечалось в статье 78, измерительная или управляющая аппаратура, содержащая значительные объемы внутренней памяти, часто подключается к компьютеру через общее с памятью адресное пространство. В этом случае аппаратуре выделяется определенный участок свободного адресного пространства из диапазона адресов 0xC0000...0xDFFFF и управление аппаратурой выполняется путем программного обращения по закрепленным за ее памятью адресам. В системах Windows NT/2000, как и в Windows 95/98, приложение работает с выделенными ей системой виртуальными адресами, а физическое адресное пространство приложению недоступно. Однако нетрудно включить в состав системы драйвер, который отобразит требуемую область физического адресного пространства на пространство виртуальных адресов и вернет в приложение виртуальный адрес начала этой области. Поскольку речь здесь идет о 32-разрядных приложениях, работающих в плоском адресном пространстве, виртуальный адрес, выделяемый в 4-гигабайтовом диапазоне адресов, совпадает с линейным. Задавая требуемые смещения относительно этого базового адреса, приложение сможет обращаться по любым адресам из отображенного диапазона. Для иллюстрации этой методики выполним отображение ПЗУ BIOS (по аналогии с примерами статей 78 и 84) и прочитаем те байты этого ПЗУ, в которых хранится дата выпуска BIOS в символьной форме. Результат работы рассматриваемого ниже примера приведен на рис. 89.1.

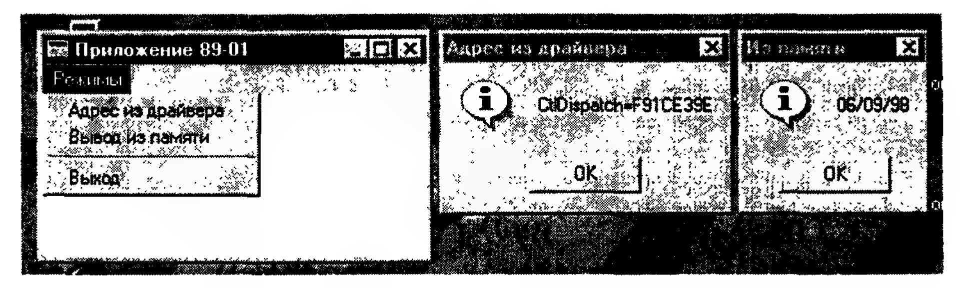


Рис. 89.1. Вывод приложения 89-01

За основу обеих программ – и приложения Windows, и драйвера – взяты программы примера 88.1. Рассмотрим сначала приложение Windows (пример 89.1).

Пример 89.1. Исходные тексты приложения Windows, выполняющего чтение из физического адресного пространства

Файл 89-01.H

```
//Определения констант
#define MI_ADDR 100
#define MI_MEM 101
#define MI_EXIT 104
#define IOCTL_ADDR (0x800<<2)|(0x22<<16)
#define IOCTL_MAP (0x801<<2)|(0x22<<16)
//Прототипы функций
void Register(HINSTANCE);
void Create(HINSTANCE);
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
void GetAddr();
void GetMem();
```

Файл 89-01.RC

```
#include "89-01.h"
Main MENU{
  POPUP "Режимы"  {
   MENUITEM "Адрес из драйвера",MI_ADDR
   MENUITEM "Вывод из памяти",MI_MEM
   MENUITEM SEPARATOR
   MENUITEM "Выход",MI_EXIT
   }
}
```

Файл 89-01.CPP

```
//Приложение 89-01. Получение содержимого физической памяти
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include "89-01.h"
char szClassName[]="MainWin";
char szTitle[]="Приложение 89-01";
HANDLE hDrv;//Дескриптор открытого драйвера
DWORD cbRet;//Счетчик байтов возвращаемых драйвером данных
PUCHAR MemAddr;//Виртуальный адрес памяти, возвращаемый драйвером
//Главная функция приложения
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE,LPSTR,int){
  MSG msg;
  Register(hInstance);
```

```
  Create(hInstance);
  while(GetMessage(&msg,NULL,0,0))DispatchMessage(&msg);
  return 0;
  }
//Функция регистрации класса главного окна
void Register(HINSTANCE hInst){
  WNDCLASS wc;
  memset(&wc,0,sizeof(wc));
  wc.lpszClassName=szClassName;
  wc.hInstance=hInst;
  wc.lpfnWndProc=WndProc;
  wc.lpszMenuName="Main";
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  RegisterClass(&wc);
  }
//Функция создания и показа главного окна
void Create(HINSTANCE hInst){
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
           10,10,250,150,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  }
//Оконная функция главного окна
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg) {
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
       return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
//Функция, вызываемая при создании главного окна
BOOL OnCreate(HWND,LPCREATESTRUCT){
  hDrv=CreateFile("\\\\.\\Win32Name",GENERIC_READ|GENERIC_WRITE,0,
                  NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
  return TRUE;
  }
//Функция, вызываемая при выборе пунктов меню
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id)  {
    case MI_ADDR://При выборе пункта "Адрес драйвера"
      GetAddr();
      break;
    case MI_MEM://При выборе пункта "Вывод из памяти"
      GetMem();
      break;
    case MI_EXIT://При выборе пункта "Выход"
      DestroyWindow(hwnd);
    }
  }
//Функция завершения приложения
void OnDestroy(HWND){
  CloseHandle(hDrv);//Закроем драйвер
  PostQuitMessage(0);
  }
//Функция получения адреса процедуры драйвера
void GetAddr(){
  ULONG DrvAddr;
  char szText[80];
```

```
//Получим адрес процедуры  драйвера CtlDispatch для отладки
  DeviceIoControl(hDrv,IOCTL_ADDR,NULL,0,&DrvAddr,4,&cbRet,NULL);
  wsprintf(szText,"CtlDispatch=%lX",DrvAddr);
  MessageBox(NULL,szText,"Адрес из драйвера",MB_ICONINFORMATION);
  }
void GetMem(){
struct{
  ULONG BaseAddr;//Базовый физический адрес памяти
  ULONG Length;//Длина отображаемого участка памяти
  }ToDriver;//Данные, пересылаемые в драйвер
  char szText[80];
  memset(szText,0,sizeof(szText));
  ToDriver.BaseAddr=0xf0000;
  ToDriver.Length=0x10000;
  DeviceIoControl(hDrv,IOCTL_MAP,&ToDriver,8,&MemAddr,4,&cbRet,NULL);
  MemAddr+=0xfff5;
  for(int i=0;i<8;i++)szText[i]=MemAddr[i];
  MessageBox(NULL,szText,"Из памяти",MB_ICONINFORMATION);
  }
```

Как видно из рис. 89.1, в меню приложения добавлен пункт "Вывод из памяти". Соответственно, в сценарий меню файла 89-01.RC включено предложение MENUITEM с идентификатором MI_MEM, а в заголовочном файле 89-01.H этому идентификатору присвоено конкретное значение 101.

В отличие от предыдущего примера, где драйвер выполнял единственную функцию передачи в приложение своих адресов, в настоящей программе предусмотрена посылка в драйвер двух различных кодов действия: одного – для получения адреса процедуры диспетчеризации (наиболее важной для отладки драйвера) и другого – для образования и возврата в приложение линейного базового адреса заданного участка памяти. В файле 89-01.H определены числовые значения этих двух кодов IOCTL_ADDR и IOCTL_MEM и, кроме того, добавлен прототип функции GetMem(), обеспечивающей интерфейс с драйвером при выборе нового пункта меню.

Исходный текст приложения изменился незначительно. В функцию OnCommand(), осуществляющую анализ выбранного пункта меню, добавлен оператор case для вызова функции GetMem() в случае выбора пункта с идентификатором MI_MEM; в текст программы включена новая функция GetMem(); несколько изменен текст функции GetAddr().

В предыдущем примере драйвер возвращал в приложение адреса всех четырех основных функций. Практически для отладки драйвера с помощью отладчика SoftICE требуется лишь адрес процедуры диспетчеризации, которая выполняет всю содержательную работу и в которой наиболее вероятны ошибки. Поэтому в настоящем варианте программы переменная DrvAddr, в которую поступают данные из драйвера, объявлена как скалярная размером в одно длинное слово, а в запросе DeviceIoControl() указана уменьшенная длина буфера – 4 байта вместо 16. Разумеется, в текст драйвера нужно будет внести соответствующие поправки, чтобы драйвер возвращал лишь одно слово, а не 4, как в примере 88.1.

Все действия по общению с физической памятью вынесены в функцию GetMem(). В ней выполняется запрос к драйверу DeviceIoControl() с кодом действия IOCTL_MEM, в котором осуществляется двухсторонний обмен информацией с драйвером. В драйвер передаются 8 байт данных из структурной переменной ToDriver: базовый адрес отображаемого участка (0xF0000) и его размер (0x10000=64 Кбайт). Драйвер возвращает единственное число длиной 4 байта – виртуальный, или линейный, адрес отображенного участка физической памяти. Этот адрес сохранится в гло-

бальной переменной MemAddr с целью дальнейшего использования в программе (в принципе не обязательно в функции GetMem()).

В реальной программе, управляющей какой-либо установкой, описанные действия следует выполнить в начале программы, на этапе инициализации, после чего все отображенное адресное пространство (в данном примере от 0xF000:0x0000 до 0xF000:0xFFFF) будет доступно для чтения и записи. Получить доступ к конкретной ячейке можно путем прибавления к базовому адресу требуемого смещения, как это сделано у нас в предложении

```
MemAddr+=0xfff5;
```

или с помощью индексной адресации, как это проиллюстрировано строчкой ниже в предложении с циклом for. В конкретном примере из ПЗУ BIOS читаются и выводятся в окно сообщения байты, хранящиеся по адресам от 0xF000:0xFFF5 до 0xF000:0xFFFC.

Рассмотрим теперь программу драйвера, обеспечивающего отображение физической памяти (пример 89.1, продолжение). Если не считать включения в программу определения добавочного кода действия, изменения коснулись лишь процедуры диспетчеризации CtlDispatch(), в которой в случае поступления из приложения кода действия IOCTL_ADDR возвращается адрес этой процедуры, а при поступлении кода IOCTL_MEM выполняются новые для нас действия по отображению памяти.

Пример 89.1 (продолжение). Драйвер,возвращающий в приложение виртуальный адрес физической памяти

```
#include "ntddk.h"
#define NT_DEVICE_NAME      L"\\Device\\NTName"//Имя объекта устройства
#define WIN32_DEVICE_NAME   L"\\??\\Win32Name"//Имя в пространстве имен Win32
//Определим коды действий при вызове драйвера
#define IOCTL_ADDR CTL_CODE \
    (FILE_DEVICE_UNKNOWN,0x800,METHOD_BUFFERED,FILE_ANY_ACCESS)
#define IOCTL_MAP CTL_CODE \
    (FILE_DEVICE_UNKNOWN,0x801,METHOD_BUFFERED,FILE_ANY_ACCESS)
//Прототипы основных функций драйвера
NTSTATUS CtlCreate(IN PDEVICE_OBJECT,IN PIRP);
NTSTATUS CtlClose(IN PDEVICE_OBJECT,IN PIRP);
NTSTATUS CtlDispatch(IN PDEVICE_OBJECT,IN PIRP);
//Функция инициализации драйвера
NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
                     IN PUNICODE_STRING RegistryPath)  {
  PDEVICE_OBJECT pDeviceObject;//Указатель на объект устройства
  UNICODE_STRING uniNtName;//Структура с NT-именем устройства
  UNICODE_STRING uniWin32Name;//Структура с Win32-именем устройства
//Преобразуем оба имени в структурные переменные со счетчиками
  RtlInitUnicodeString(&uniNtName,NT_DEVICE_NAME );
  RtlInitUnicodeString(&uniWin32Name,WIN32_DEVICE_NAME );
//Создадим символическую связь NT- и Win32-имен
  IoCreateSymbolicLink(&uniWin32Name,&uniNtName);
//Создадим объект устройства
  IoCreateDevice(pDriverObject,0,&uniNtName,FILE_DEVICE_UNKNOWN,
             0,FALSE,&pDeviceObject);
//Заполним в структуре объекта драйвера поля с адресами основных функций
  pDriverObject->MajorFunction[IRP_MJ_CREATE]=CtlCreate;
  pDriverObject->MajorFunction[IRP_MJ_CLOSE]=CtlClose;
  pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]=CtlDispatch;
  return STATUS_SUCCESS;
  }
//Функция, вызываемая при открытии драйвера приложением
NTSTATUS CtlCreate(IN PDEVICE_OBJECT pDeviceObject,IN PIRP Irp)  {
```

```
  Irp->IoStatus.Status=STATUS_SUCCESS;
  Irp->IoStatus.Information=0;
  IoCompleteRequest(Irp,IO_NO_INCREMENT );
  return STATUS_SUCCESS;
  }
//Функция, вызываемая при закрытии драйвера приложением
NTSTATUS CtlClose(IN PDEVICE_OBJECT pDeviceObject,IN PIRP Irp)  {
  Irp->IoStatus.Status=STATUS_SUCCESS;
  Irp->IoStatus.Information=0;
  IoCompleteRequest(Irp,IO_NO_INCREMENT );
  return STATUS_SUCCESS;
  }
//Функция диспетчеризации
NTSTATUS CtlDispatch(IN PDEVICE_OBJECT pDeviceObject,IN PIRP Irp)  {
  PIO_STACK_LOCATION pIrpStack;//Указатель на стековую область
  PULONG pIOBuffer;//Системный буфер обмена данными с приложением
  UNICODE_STRING uniMemoryName;//Имя объекта "секция памяти"
  OBJECT_ATTRIBUTES ObjectAttributes;//Атрибуты объекта
  HANDLE SectionHandle;//Дескриптор секции памяти
  PHYSICAL_ADDRESS PhysicalAddress;//Физический адрес
  PHYSICAL_ADDRESS SectionBase;//То же для коррекции
  ULONG Length;//Размер отображаемого участка
  PVOID VirtualAddress=NULL;//Возвращаемый виртуальный адрес
  pIrpStack=IoGetCurrentIrpStackLocation(Irp);//Получим адрес стековой области
  pIOBuffer=Irp->AssociatedIrp.SystemBuffer;//Получим адрес буфера обмена
  switch (pIrpStack->Parameters.DeviceIoControl.IoControlCode) {
    case IOCTL_ADDR://Код первого действия
      *pIOBuffer=(ULONG)CtlDispatch;//Вернем адрес этой процедуры
      Irp->IoStatus.Information=4;//Число пересылаемых байтов
      break;
    case IOCTL_MAP//Код второго действия
//Заберем из буфера обмена оба параметра - базовый адрес и размер отображения
      PhysicalAddress.LowPart=*pIOBuffer++;
      PhysicalAddress.HighPart=0;
      Length=*pIOBuffer--;
//Создадим имя объекта секции памяти
      RtlInitUnicodeString(&uniMemoryName,L"\\Device\\PhysicalMemory");
//Подготовим атрибуты объекта для ZwOpenXxx()
      InitializeObjectAttributes(&ObjectAttributes,
           &uniMemoryName,OBJ_CASE_INSENSITIVE,NULL,NULL);
//Создадим объект секции памяти
      ZwOpenSection(&SectionHandle,SECTION_ALL_ACCESS,&ObjectAttributes);
//Отобразим пространство адресов секции на пространство виртуальных адресов.
      SectionBase=PhysicalAddress;//Чтобы не потерять заданную нами базу
      ZwMapViewOfSection(SectionHandle,(HANDLE)-1,&VirtualAddress,
          0L,Length,&SectionBase,&Length,ViewShare,0,
          PAGE_READWRITE|PAGE_NOCACHE);
//Скорректируем возвращенный виртуальный адрес
      (ULONG)VirtualAddress+=PhysicalAddress.LowPart-SectionBase.LowPart;
      *pIOBuffer=(ULONG)VirtualAddress;//Отправим адрес в приложение
      Irp->IoStatus.Information=4;//Число пересылаемых байтов
      break;
    }
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest( Irp, IO_NO_INCREMENT );
    return STATUS_SUCCESS;
}
```

В функции диспетчеризации, помимо уже знакомых нам переменных для указателя на стековую область IRP, буфера обмена и др., вводится целый ряд новых

переменных, назначение которых будет ясно из последующего обсуждения. Отметим здесь, что структурный тип PHYSICAL_ADDRESS (или эквивалентный ему тип LARGE_INTEGER) представляет собой комбинацию из двух длинных слов:

```
struct PHYSICAL_ADDRESS {
  ULONG LowPart;
  LONG HighPart;
  };
```

Переменные этого типа требуется использовать в качестве параметров при вызове некоторых функций, связанных с отображением адресного пространства.

Рассмотрим фрагмент драйвера, выполняемый при получении кода действия IOCTL_MEM.

Прежде всего из буфера обмена извлекается первое помещенное туда приложением данное – базовый физический адрес отображаемого участка. Эта величина помещается в младшую часть переменной PhysicalAddress типа PHYSICAL_ADDRESS; старшая часть этой переменной обнуляется. Использование постинкрементной адресации (символ ++) приводит к автоматической настройке указателя pIOBuffer на адрес следующего данного в буфере обмена. Далее из буфера извлекается это следующее данное – размер отображаемого участка адресов. Постдекрементная адресация (символ --) возвращает указатель pIOBuffer в исходное состояние. Это в данном случае существенно, так как в дальнейшем мы будем пользоваться тем же указателем (и разумеется, тем же буфером) для передачи данных назад в приложение. Очевидно, что засылать передаваемые в приложение данные в буфер следует от его начала.

Для адресации физической памяти следует создать специальный объект – секцию памяти. Этот объект, как и, скажем, объект драйвера, должен иметь свое имя (в формате структурной переменной со счетчиком), которое создается вызовом уже знакомой нам функции RtlInitUnicodeString(). Исходное имя в виде строки Unicode (L"\\Device\\PhysicalMemory") подставлено непосредственно в вызов функции в качестве второго параметра.

Функция создания объекта памяти требует в качестве одного из своих параметров адрес структуры типа OBJECT_ATTRIBUTES с атрибутами создаваемого объекта. Для инициализации этой структуры (переменная ObjectAttributes) используется функция InitializeObjectAttributes(), принимающая в качестве выходного параметра адрес этой переменной, а в качестве входных – адрес строки с именем объекта, а также ряд констант, назначение которых можно посмотреть в справочнике DDK NT.

Объект секции памяти создается с помощью функции ZwOpenSection(), которая возвращает дескриптор объекта по адресу своего первого параметра. Хотя секция памяти создана, она имеет пока, так сказать, обезличенный вид, так как не закреплена ни за какими конкретными адресами. Собственно отображение физических адресов на виртуальные осуществляется вызовом функции ZwMapViewOfSection(), которой среди прочих параметров передаются дескриптор объекта секции памяти, дескриптор текущего процесса (числовое значение –1, преобразованное в тип HANDLE), адрес ячейки для помещения результата отображения (VirtualAddress), длина отображаемого участка (Length), еще раз адрес переменной Length и, наконец, адрес переменной с базовым адресом отображаемого участка. Необходимо заметить, что начальное

значение виртуального адреса перед вызовом функции ZwMapViewOfSection() должно быть равно NULL.

Для конкретных данных нашего примера, где базовый адрес кратен 64 Кбайт, функция ZwMapViewOfSection() вернет соответствующий виртуальный адрес (например, 0x00C90000), который можно переслать в приложение и использовать там для работы с физической памятью; никаких дополнительных операций с этим адресом выполнять не нужно.

Однако в тех случаях, когда базовый адрес не кратен 64 Кбайт (например, мы указали в качестве базового адрес 0xF0010), возникает осложнение. Дело в том, что функция ZwMapViewOfSection() и в этом случае отобразит на виртуальные адреса участок физической памяти, начинающийся с адреса, ближайшего к указанному, но кратного 64 Кбайт (т. е. того же адреса 0xF0000). Для того чтобы скомпенсировать возможный сдвиг, следует сместить полученный виртуальный адрес на ту же величину, на которую оказался смещенным адрес базовый. Функция ZwMapViewOfSection() возвращает через 6-й параметр (&SectionBase) тот физический адрес, который она в действительности приняла в качестве базового. Этот параметр, таким образом, выступает в качестве как входного, так и выходного. Перед вызовом функции ZwMapViewOfSection() в ячейке SectionBase находится заданный нами физический адрес (например, 0xF0010), а после выполнения этой функции – использованный ею базовый адрес (0xF0000 для нашего примера).

Для внесения необходимой коррекции адреса перед вызовом функции отображения ZwMapViewOfSection() задаваемый нами базовый адрес копируется из ячейки PhysicalAddress в ячейку SectionBase и адрес этой ячейки передается функции ZwMapViewOfSection(). Затем вычисляется разность содержимого младших половин ячеек PhysicalAddress (где остался наш адрес) и SectionBase (куда функция отображения вернула использованный ею и, возможно, округленный адрес), и полученная разность прибавляется к "неправильному" виртуальному адресу, возвращенному функцией отображения. Этот скорректированный адрес и возвращается в приложение.

Статья 90. **Драйвер для управления аппаратурой через порты**

Прямое обращение из приложения к портам аппаратуры в системах Windows NT запрещено, и для управления такой аппаратурой следует разработать соответствующий драйвер. В зависимости от конкретных потребностей на драйвер можно возложить более или менее комплексные функции; здесь мы рассмотрим драйвер, выполняющий по запросам приложения элементарные операции чтения-записи одного байта через указанный в вызове порт. Воспользуемся в качестве программируемой установки экспериментальной платой счетчика-таймера, описанной в статье 80. Поскольку приложение Windows, осуществляющее полный контроль работы установки, оказывается довольно громоздким, а драйвер, реализующий обращение к портам, наоборот, весьма простым, рассмотрим сначала программу драйвера (пример 90.1).

Пример 90.1. Драйвер, обеспечивающий элементарные операции по вводу-выводу через порты

```
#include "ntddk.h"
#define NT_DEVICE_NAME      L"\\Device\\NTName"//Имя объекта устройства
#define WIN32_DEVICE_NAME   L"\\??\\Win32Name"//Имя в пространстве имен Win32
//Определим коды действий при вызове драйвера
#define IOCTL_ADDR CTL_CODE \
    (FILE_DEVICE_UNKNOWN,0x800,METHOD_BUFFERED,FILE_ANY_ACCESS)
#define IOCTL_READ CTL_CODE \
    (FILE_DEVICE_UNKNOWN,0x801,METHOD_BUFFERED,FILE_ANY_ACCESS)
#define IOCTL_WRITE CTL_CODE \
    (FILE_DEVICE_UNKNOWN,0x802,METHOD_BUFFERED,FILE_ANY_ACCESS)
//Прототипы основных функций драйвера
NTSTATUS CtlCreate(IN PDEVICE_OBJECT,IN PIRP);
NTSTATUS CtlClose(IN PDEVICE_OBJECT,IN PIRP);
NTSTATUS CtlDispatch(IN PDEVICE_OBJECT,IN PIRP);
//Функция инициализации драйвера
NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
                     IN PUNICODE_STRING RegistryPath)  {
  PDEVICE_OBJECT pDeviceObject;//Указатель на объект устройства
  UNICODE_STRING uniNtName;//Структура с именем устройства;
  UNICODE_STRING uniWin32Name;//Структура с Win32-именем устройства ing;
//Преобразуем оба имени в структурные переменные со счетчиками
  RtlInitUnicodeString(&uniNtName,NT_DEVICE_NAME);
  RtlInitUnicodeString(&uniWin32Name,WIN32_DEVICE_NAME);
//Создадим символическую связь NT- и Win32-имен
  IoCreateSymbolicLink(&uniWin32Name,&uniNtName);
//Создадим объект устройства
  IoCreateDevice(pDriverObject,0,&uniNtName,FILE_DEVICE_UNKNOWN,
             0,FALSE,&pDeviceObject);
//Заполним в структуре объекта драйвера поля с адресами основных функций
  pDriverObject->MajorFunction[IRP_MJ_CREATE]=CtlCreate;
  pDriverObject->MajorFunction[IRP_MJ_CLOSE]=CtlClose;
  pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]=CtlDispatch;
  return STATUS_SUCCESS;
  }
//Функция, вызываемая при открытии драйвера приложением
NTSTATUS CtlCreate(IN PDEVICE_OBJECT pDeviceObject,IN PIRP Irp){
  Irp->IoStatus.Status=STATUS_SUCCESS;
  Irp->IoStatus.Information=0;
  IoCompleteRequest(Irp,IO_NO_INCREMENT);
  return STATUS_SUCCESS;
  }
//Функция, вызываемая при закрытии драйвера приложением
NTSTATUS CtlClose(IN PDEVICE_OBJECT pDeviceObject,IN PIRP Irp){
  Irp->IoStatus.Status=STATUS_SUCCESS;
  Irp->IoStatus.Information=0;
  IoCompleteRequest(Irp,IO_NO_INCREMENT);
  return STATUS_SUCCESS;
  }
//Функция диспетчеризации
NTSTATUS CtlDispatch(IN PDEVICE_OBJECT pDeviceObject,IN PIRP Irp){
  PIO_STACK_LOCATION pIrpStack;//Указатель на стековую область
  PVOID pIOBuffer;//Системный буфер обмена данными с приложением
  USHORT Port;//Номер порта, получаемый из приложения
  pIrpStack=IoGetCurrentIrpStackLocation(Irp);//Получим адрес стековой области
  pIOBuffer=Irp->AssociatedIrp.SystemBuffer;//Получим адрес буфера обмена
  switch (pIrpStack->Parameters.DeviceIoControl.IoControlCode){
    case IOCTL_ADDR://Код 1-го действия - пересылка в приложение адреса
      *(PULONG)pIOBuffer=(ULONG)CtlDispatch;//Перешлем адрес CtlDispatch
      Irp->IoStatus.Information=4;//Пересылаем 4 байта
```

```
        break;
    case IOCTL_READ://Код 2-го действия - ввод байта из заданного порта
        Port=*(PUSHORT)pIOBuffer;//Получим номер порта
        *((PUCHAR)pIOBuffer)=READ_PORT_UCHAR((PUCHAR)Port);//Ввод и пересылка
        Irp->IoStatus.Information=1;//Пересылаем 1 байт
        break;
    case IOCTL_WRITE://Код 3-го действия - вывод заданного байта в заданный порт
        Port=*((PUSHORT)pIOBuffer)++;//Получим номер порта
        WRITE_PORT_UCHAR((PUCHAR)Port,*(PUCHAR)pIOBuffer);//Вывод в порт
        Irp->IoStatus.Information=0;//В приложение не пересылаем ничего
        break;
    }
    Irp->IoStatus.Status=STATUS_SUCCESS;
    IoCompleteRequest(Irp,IO_NO_INCREMENT);
    return STATUS_SUCCESS;
}
```

В драйвере определены три кода действия: IOCTL_ADDR – для пересылки в приложение адреса функции диспетчеризации, IOCTL_READ – для чтения одного байта из заданного в запросе приложения порта и IOCTL_WRITE – для записи в заданный в запросе порт заданного там же байта данных.

В функции диспетчеризации CtlDispatch() помимо уже известных нам данных определена короткая целочисленная переменная без знака Port, в которую из приложения будет поступать номер текущего порта. Имеется некоторая особенность в объявлении переменной pIOBuffer. В предыдущих примерах через буфер обмена передавались только длинные слова и переменной pIOBuffer можно было назначить тип PULONG (указатель на переменную типа ILONG). В настоящем примере буфер обмена будет служить для передачи как коротких слов (номер порта), так и байтов (пересылаемые данные). Поэтому переменной pIOBuffer назначен обобщенный тип PVOID (указатель на что угодно), который по мере необходимости можно будет преобразовывать в требуемый тип. Так, в предложении

```
Port=*((PUSHORT)pIOBuffer)++;//Получим номер порта
```

переменная pIOBuffer рассматривается как указатель на короткое слово, в результате чего операция постинкремента увеличивает ее значение на 2 – размер короткого слова, а в предложении

```
*((PUCHAR)pIOBuffer)=READ_PORT_UCHAR((PUCHAR)Port);//Ввод и пересылка
```

эта же переменная рассматривается как указатель на байт (PUCHAR), что соответствует типу значения, возвращаемого функцией READ_PORT_UCHAR(). Между прочим, сама эта переменная, будучи указателем, во всех случаях занимает в 32-разрядной программе длинное слово.

При поступлении в драйвер кода действия CTL_READ из системного буфера обмена в переменную драйвера Port читается короткое слово без знака. Для прямого обращения к порту используется функция ядра Windows READ_PORT_UCHAR(), требующая в качестве единственного параметра типа PUCHAR (указатель на байт без знака) номер порта. Результат выполнения этой функции помещается в буфер обмена, которому для этой операции назначается тот же тип PUCHAR. В переменную Irp->IoStatus.Information помещается 1 – число пересылаемых в приложение байтов.

При поступлении в драйвер кода действия CTL_WRITE из системного буфера обмена точно так же читается номер порта, но адрес буфера при этом инкрементируется, чтобы получить указатель на следующее данное в буфере – записываемый в

порт байт. Далее используется функция ядра Windows WRITE_PORT_UCHAR(), требующая в качестве параметров номер порта и записываемое в порт данное. Ради наглядности следовало бы сначала извлечь данное из буфера обмена, а затем указать это данное в виде второго параметра функции WRITE_PORT_UCHAR(). В программе драйвера эти действия объединены. Кстати, и в ячейке для номера порта особой необходимости не было. Операцию чтения номера порта из буфера обмена можно было включить прямо в обозначение первого параметра функции WRITE_PORT_UCHAR():

```
WRITE_PORT_UCHAR((PUCHAR)(*((PUSHORT)pIOBuffer)++),*(PUCHAR)pIOBuffer);
```

Однако такое нагромождение операций, типичное для программ на языке Си, вряд ли будет способствовать пониманию сути дела.

Операция записи в порт не требует пересылки в приложение каких-либо данных, поэтому в буфер обмена ничего на записывается, а переменной Irp->IoStatus. Information придается значение 0.

Рассмотрим теперь приложение Windows, осуществляющее управление платой (пример 90.1, продолжение). Поскольку оно в принципе не отличается от предыдущих, мы опустили тексты файлов заголовочного и ресурсов, а текст программы привели в сокращенном виде. На рис. 90.1 показано меню приложения и образец его вывода при настроечных константах, использованных в примере.

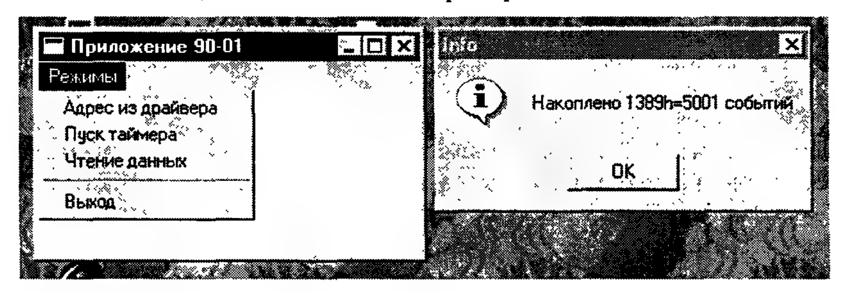


Рис. 90.1. Главное окно и вывод приложения 90-01

Пример 90.1 (продолжение). Приложение Windows, осуществляющее управление платой счетчика-таймера.

```
#include <windows.h>
#include <windowsx.h>
#include "90-01.h"
char szClassName[]="MainWin";
char szTitle[]="Приложение 90-01";
HANDLE hDrv;//Дескриптор открытого драйвера
DWORD cbRet;//Счетчик байтов возвращаемых драйвером данных
UINT DrvAddr;
WORD Constant,Result;
struct {
USHORT Port;
UCHAR Value;
}Data;
//Главная функция приложения
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE,LPSTR,int){
...
  }
//Функция регистрации класса главного окна
```

```
void Register(HINSTANCE hInst){
...
  }
//Функция создания и показа главного окна
void Create(HINSTANCE hInst){
...
  }
//Оконная функция главного окна
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
...
  }
//Функция, вызываемая при создании главного окна
BOOL OnCreate(HWND,LPCREATESTRUCT){
  hDrv=CreateFile("\\\\.\\Win32Name",GENERIC_READ|GENERIC_WRITE,0,
                  NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
  return TRUE;
  }
//Функция, вызываемая при выборе пунктов меню
void OnCommand(HWND hwnd,int id,HWND,UINT){
  switch(id)  {
    case MI_ADDR://При выборе пункта "Адрес из драйвера"
      GetAddr();
      break;
    case MI_START://При выборе пункта "Пуск таймера"
      StartTimer();
      break;
    case MI_DATA://При выборе пункта "Чтение данных"
      GetData();
      break;
    case MI_EXIT://При выборе пункта "Выход"
      DestroyWindow(hwnd);
    }
  }
//Функция завершения приложения
void OnDestroy(HWND){
...
  }
//Функция получения адреса процедуры CtlDispatch
void GetAddr(){
...
  }
//Функция инициализации таймера и пуска измерений
void StartTimer(){
//Инициализируем  таймер
    Data.Port=0x30c;
    DeviceIoControl(hDrv,IOCTL_READ,&Data,3,&Data.Value,1,&cbRet,NULL);
    Data.Port=0x303;
    Data.Value=0x36;
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Value=0x70;
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Value=0xb6;
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
//Канал 0
    Constant=100;
    Data.Port=0x300;
    Data.Value=LOBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Value=HIBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
//Канал 1
```

```
    Constant=50000;
    Data.Port=0x301;
    Data.Value=LOBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Value=HIBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
//Инициализируем внутренний генератор
    Data.Port=0x302;
    Constant=1000;
    Data.Value=LOBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Value=HIBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
//Включаем счет
    Data.Port=0x30b;
    DeviceIoControl(hDrv,IOCTL_READ,&Data.Port,3,&Data.Value,1,&cbRet,NULL);
    }
//Функция чтения данных
void GetData(){
    char szText[80];
    Data.Port=0x309;//Старший байт результата
    DeviceIoControl(hDrv,IOCTL_READ,&Data.Port,3,&Data.Value,1,&cbRet,NULL);
    Result=(WORD)Data.Value;
    Result<<=8;
    Data.Port=0x308;//Младший байт результата
    DeviceIoControl(hDrv,IOCTL_READ,&Data.Port,3,&Data.Value,1,&cbRet,NULL);
    Result|=(WORD)Data.Value;
    wsprintf(szText,"Накоплено %Xh=%d событий",Result,Result);
    MessageBox(NULL,szText,"Info",MB_ICONINFORMATION);
    }
```

Фактически оба интересующих нас фрагмента – инициализация платы счетчика-таймера и чтение накопленных данных – представляют собой последовательности вызовов DeviceIoControl() с указанием того или иного кода действия и соответствующих параметров. Операции деления короткого слова на байты для посылки в драйвер и через него в порты рассматривались в предыдущих статьях. Обратная операция – объединение двух байтов, прочитанных из порта (в функции GetData()), также уже встречалась в статье 80.

Статья 91. **Драйверы для обслуживания аппаратных прерываний**

В системах Windows NT, так же как и в Windows 95/98, обработка аппаратных прерываний требует решения в программном комплексе драйвер-приложение следующих задач:

- включения в состав драйвера обработчика аппаратного прерывания;
- передачи данных из обработчика прерываний в приложение;
- оповещения приложения о регистрации драйвером прерывания, т. е., по существу, включения в приложение "приложенческого" обработчика прерываний, который будет активизироваться "драйверным" обработчиком.

Общие принципы обработки прерываний в 32-разрядных приложениях Windows, а также связанные с этим понятия (процедуры прерываний и отложенных прерываний, потоки и события и др.) последовательно рассматривались в статьях 86 и 87; здесь мы

сразу рассмотрим программный комплекс, объединяющий решение всех перечисленных выше задач. В качестве примера программируемой аппаратуры использована плата счетчика-таймера. В меню приложения включены команды диагностического вывода на экран характерных адресов драйвера и пуска измерений (рис. 91.1). Команда чтения результата в данном случае не нужна, так как операцию вывода накопленных данных естественно включить в обработчик прерываний приложения, который активизируется автоматически по завершении измерений.

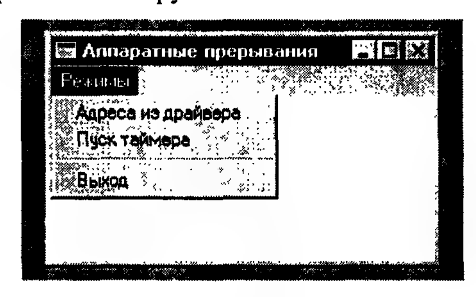


Рис. 91.1. Меню приложения 91-01

При выборе пункта меню "Пуск таймера" приложение с помощью последовательности запросов ввода-вывода инициализирует таймер (в точности так же, как и в предыдущем примере) и запускает процесс накопления событий. По истечении заданного интервала времени аппаратное прерывание уровня IRQ5 активизирует обработчик прерываний, включенный в программу драйвера. В рассматриваемом примере обработчик прерывания не выполняет содержательной работы, а только создает объект сброшенного события и ставит запрос на выполнение процедуры отложенной обработки прерывания. Назначение этой процедуры – установка сброшенного ранее события, приводящая к активизации в приложении потока, включающего в себя обработчик прерываний приложения. Этот обработчик с помощью последовательности запросов ввода-вывода считывает число событий, накопленных в счетчике платы, и выводит его в окно сообщения.

Рассмотрим сначала программу драйвера (пример 91.1).

Пример 91.1. Драйвер, обслуживающий аппаратные прерывания

```
#include "ntddk.h"
#define NT_DEVICE_NAME      L"\\Device\\NTName"//Имя объекта устройства
#define WIN32_DEVICE_NAME   L"\\??\\Win32Name"//Имя в пространстве имен Win32
//Определим коды действий при вызове драйвера
#define IOCTL_ADDR CTL_CODE \
    (FILE_DEVICE_UNKNOWN,0x800,METHOD_BUFFERED,FILE_ANY_ACCESS)
#define IOCTL_READ CTL_CODE \
    (FILE_DEVICE_UNKNOWN,0x801,METHOD_BUFFERED,FILE_ANY_ACCESS)
#define IOCTL_WRITE CTL_CODE \
    (FILE_DEVICE_UNKNOWN,0x802,METHOD_BUFFERED,FILE_ANY_ACCESS)
#define IOCTL_START CTL_CODE \
    (FILE_DEVICE_UNKNOWN,0x803,METHOD_BUFFERED,FILE_ANY_ACCESS)
#define CARD_IRQ 5//Аппаратный уровень прерываний
//Прототипы основных функций драйвера
NTSTATUS CtlCreate(IN PDEVICE_OBJECT,IN PIRP);
NTSTATUS CtlClose(IN PDEVICE_OBJECT,IN PIRP);
NTSTATUS CtlDispatch(IN PDEVICE_OBJECT,IN PIRP);
//Обработчик прерываний драйвера
BOOLEAN IsrRoutine(IN PKINTERRUPT,IN OUT PVOID);
//Процедура отложенных прерываний
```

```
VOID DpcRoutine(IN PKDPC,IN PDEVICE_OBJECT,IN PIRP,IN PVOID);
//Структура, описывающая состав расширения устройства
typedef struct _EXTENSION {
  PKEVENT pEventObject;//Объект события
  PIRP pIrp;//Пакет ввода-вывода
  } EXTENSION,*PEXTENSION;
//Процедура инициализации драйвера
NTSTATUS DriverEntry(IN PDRIVER_OBJECT pDriverObject,
                     IN PUNICODE_STRING RegistryPath)  {
  PDEVICE_OBJECT pDeviceObject;//Указатель на объект устройства
  UNICODE_STRING uniNtName;//Структура с NT-именем устройства
  UNICODE_STRING uniWin32Name;//Структура с Win32-именем устройства
  UNICODE_STRING EventName;//Структура с именем события
  HANDLE EventHandle;//Дескриптор события
  PEXTENSION pExt; //Расширение устройства
  KIRQL irql=CARD_IRQ; //Аппаратный уровень используемых прерываний
  KAFFINITY Affinity;//Сродство
  PKINTERRUPT pInterruptObject;//Объект прерывания
  ULONG MappedVector;//Аппаратный вектор платы в NT
//Преобразуем оба имени в структурные переменные со счетчиками
  RtlInitUnicodeString( &uniNtName, NT_DEVICE_NAME );
  RtlInitUnicodeString(&uniWin32Name,WIN32_DEVICE_NAME);
//Создадим символическую связь NT- и Win32-имен
  IoCreateSymbolicLink(&uniWin32Name,&uniNtName);
//Создадим объект устройства
  IoCreateDevice(pDriverObject,sizeof(EXTENSION),&uniNtName,
           FILE_DEVICE_UNKNOWN,0,FALSE,&pDeviceObject);
//Заполним в структуре объекта драйвера поля с адресами основных функций
  pDriverObject->MajorFunction[IRP_MJ_CREATE]=CtlCreate;
  pDriverObject->MajorFunction[IRP_MJ_CLOSE]=CtlClose;
  pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]=CtlDispatch;
//Очистим расширение устройства
  RtlZeroMemory(pDeviceObject->DeviceExtension,sizeof(EXTENSION));
  pExt=pDeviceObject->DeviceExtension;
//Инициализация прерываний
//Получим отображенный вектор
  MappedVecor=HalGetInterruptVector(Isa,0,CARD_IRQ,CARD_IRQ,&irql,&Affinity);
//Зарегистрируем обработчик прерываний драйвера
  IoConnectInterrupt(&pInterruptObject,IsrRoutine,pDeviceObject,NULL,
        MappedVector,irql,irql,Latched,FALSE,Affinity,FALSE);
//Инициализация отложенного вызова и связи с приложением
//Зарегистрируем процедуру отложенного вызова
  IoInitializeDpcRequest(pDeviceObject,DpcRoutine);
//Образуем структуру со счетчиком для имени события
  RtlInitUnicodeString(&EventName,L"\\BaseNamedObjects\\SignalEvent");
//Создадим событие для оповещения приложения
  pExt->pEventObject=IoCreateNotificationEvent(&EventName,&EventHandle);
  KeClearEvent(pExt->pEventObject);//Сбросим событие, запретив выполнение
  return STATUS_SUCCESS;
  }
//Функция, вызываемая при открытии драйвера приложением
NTSTATUS CtlCreate(IN PDEVICE_OBJECT pDeviceObject,IN PIRP pIrp){
  pIrp->IoStatus.Status=STATUS_SUCCESS;
  pIrp->IoStatus.Information=0;
  IoCompleteRequest(pIrp,IO_NO_INCREMENT );
  return STATUS_SUCCESS;
  }
//Функция, вызываемая при закрытии драйвера приложением
NTSTATUS CtlClose(IN PDEVICE_OBJECT pDeviceObject,IN PIRP pIrp) {
  pIrp->IoStatus.Status=STATUS_SUCCESS;
  pIrp->IoStatus.Information=0;
```

```
  IoCompleteRequest(pIrp,IO_NO_INCREMENT );
  return STATUS_SUCCESS;
  }
//Функция диспетчеризации
NTSTATUS CtlDispatch(IN PDEVICE_OBJECT pDeviceObject,IN PIRP pIrp){
  PIO_STACK_LOCATION pIrpStack;//Указатель на стековую область
  PVOID pIOBuffer;//Системный буфер обмена данными с приложением
  USHORT Port;//Номер порта, получаемый из приложения
  PEXTENSION pExt;//Указатель на расширение устройства
  pExt=pDeviceObject->DeviceExtension;//Инициализация указателя
  pIrpStack=IoGetCurrentIrpStackLocation(pIrp);//Получим адрес стековой области
  pIOBuffer=pIrp->AssociatedIrp.SystemBuffer;//Получим адрес буфера обмена
  switch (pIrpStack->Parameters.DeviceIoControl.IoControlCode)    {
    case IOCTL_ADDR://Код 1-го действия - пересылка в приложение адресов
      *((PULONG)pIOBuffer)++=(ULONG)CtlDispatch;
      *((PULONG)pIOBuffer)++=(ULONG)IsrRoutine;
      *((PULONG)pIOBuffer)=(ULONG)DpcRoutine;
      pIrp->IoStatus.Information=12;//Пересылаем 12 байт
      break;
    case IOCTL_READ://Код 2-го действия - ввод байта из заданного порта
      Port=*(PUSHORT)pIOBuffer;//Получим номер порта
      *((PUCHAR)pIOBuffer)=READ_PORT_UCHAR((PUCHAR)Port);//Ввод и пересылка
      pIrp->IoStatus.Information=1;//Пересылаем 1 байт
      break;
    case IOCTL_WRITE://Код 3-го действия - вывод заданного байта в заданный порт
      Port=*((PUSHORT)pIOBuffer)++;//Получим номер порта
      WRITE_PORT_UCHAR((PUCHAR)(Port),*(PUCHAR)pIOBuffer);//Вывод в порт
      pIrp->IoStatus.Information=0;//В приложение не пересылаем ничего
      break;
    case IOCTL_START://Код 4-го действия - создание отложенного IRP
      pExt->pIrp=pIrp;//Сохраним адрес пакета ввода-вывода в расширении
      pIrp->IoStatus.Status=STATUS_SUCCESS;
      pIrp->IoStatus.Information=0;
      IoMarkIrpPending(pIrp);//Отметим IRP отложенным
      return STATUS_SUCCESS;
    }
    pIrp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(pIrp, IO_NO_INCREMENT );
    return STATUS_SUCCESS;
  }
//Процедура обработки прерываний
BOOLEAN IsrRoutine(IN PKINTERRUPT pInterrupt,IN PVOID pContext){
  PDEVICE_OBJECT pDeviceObject=pContext;//Получим наш объект устройства
  PEXTENSION pExt=pDeviceObject->DeviceExtension;//Получим адрес расширения
  IoRequestDpc(pDeviceObject,pExt->pIrp,NULL);//Поставим в DPC очередь
  return TRUE;
  }
//Процедура отложенной обработки прерываний
VOID DpcRoutine(IN PKDPC Dpc,IN PDEVICE_OBJECT pDeviceObject,
                IN PIRP pIrp,IN PVOID pContext){
  PEXTENSION pExt=pDeviceObject->DeviceExtension;//Получим адрес расширения
  KeSetEvent(pExt->pEventObject,0,FALSE);//Установим событие, разрешив выполнение
  KeClearEvent(pExt->pEventObject);//Сбросим событие
  pIrp->IoStatus.Information=0;
  pIrp->IoStatus.Status=STATUS_SUCCESS;
  IoCompleteRequest(pIrp,IO_NO_INCREMENT);
  }
```

Особенности обработки аппаратных прерываний заставили увеличить число кодов действий. Дополнительный код действия IOCTL_START служит для организации пе-

редачи пакета запроса ввода-вывода в процедуру отложенной обработки прерываний. В дальнейшем этот вопрос будет рассмотрен подробнее.

Существенным отличием данного драйвера от предыдущих является использование расширения устройства (см. рис. 88.3) – области данных произвольного объема для реализации обмена данными между отдельными запросами ввода-вывода. Состав расширения устройства целиком определяется составителем драйвера; в рассматриваемом примере расширение оформлено в виде структуры, типу которой дано произвольное имя _EXTENSION. С помощью оператора определения типа typedef введено альтернативное обозначение того же типа EXTENSION, а также имя PEXTENSION для обозначения указателя на структурные переменные данного типа. В нашем случае расширение имеет небольшой размер и содержит всего две переменные – указатели на объект события и на пакет ввода-вывода. Обе эти переменные надо будет передать в обработчик прерываний драйвера, а оттуда – в процедуру обработки отложенных прерываний. В других случаях, когда некоторые данные должны передаваться от одного запроса ввода-вывода к другому (например, в результате первого запроса данные читаются из устройства и сохраняются в драйвере, а в результате второго они передаются в приложение), эти данные так же должны быть помещены в расширение устройства.

В процедуре инициализации драйвера используется целый ряд дополнительных данных: имя события EventName и дескриптор события EventHandle, указатель pExt на структурную переменную типа EXTENSION, указатель pInterruptObject на объект прерывания и др. Со всеми этими переменными мы еще столкнемся при описании программы драйвера.

В начале процедуры инициализации выполняются те же действия, что и в предыдущих примерах: преобразование обоих имен устройства в структурные переменные со счетчиками и создание между ними символической связи, создание объекта устройства, заполнение полей в объекте драйвера адресами основных функций. Обратите внимание на вызов функции IoCreateDevice(), создающей объект устройства: в качестве второго параметра указан размер планируемого нами расширения устройства. Отличие этого параметра от нуля заставляет диспетчер ввода-вывода при создании объекта устройства включить в его состав расширение заданного нами размера. Для того чтобы избежать появления в расширении устройства случайных значений ("мусора"), оно с помощью функции RtlZeroMemory заполняется нулями.

Как видно из рис. 88.3, указатель на расширение (с именем DeviceExtension) входит в объект устройства. С другой стороны, указатель на объект устройства передается в драйвер из Windows в качестве первого параметра процедуры инициализации, что дает возможность получить доступ к расширению. Для удобства дальнейших обращений к расширению значение указателя на него переносится в локальную переменную pExt. Необходимо иметь в виду, что эта переменная действительна лишь в процессе выполнения функции DriverEntry(); в других функциях драйвера, активизируемых запросами ввода-вывода, указатель на расширение устройства придется каждый раз получать заново.

Как уже говорилось ранее (см. статью 80), системы Windows, работающие в защищенном режиме, вынуждены в процессе загрузки перепрограммировать контроллеры прерываний компьютера, так как векторы аппаратных прерываний с 8-го по 15-й, используемые DOS, в защищенном режиме принадлежат исключениям процессора.

В Windows 95/98 под аппаратные прерывания отводятся векторы 0x50...0x5F; в Windows NT аппаратные векторы расположены в диапазоне 0x30...0x3F.

Как известно, в DOS аппаратные прерывания обладают относительными приоритетами, совпадающими с номерами уровней запросов прерываний: от 0 для первого уровня ведущего контроллера IRQ0, к которому подключен системный таймер (наивысший приоритет), до 15 для последнего уровня IRQ15 ведомого контроллера (низший приоритет).

В Windows NT используется другая, значительно более сложная система уровней приоритетов. Каждый активизированный поток обладает определенным уровнем приоритета, при этом чем больше значение уровня, тем приоритет потока выше. Система предоставляет процессорное время только потокам с наивысшим текущим приоритетом, выделяя по очереди каждому такому потоку определенный квант времени, что и обеспечивает параллельное выполнение многих задач. Потоков с высоким приоритетом в системе может не быть, или эти потоки могут приостановить свое выполнение, например в ожидании ввода с клавиатуры или сигнала от мыши. В этом случае кванты процессорного времени отдаются потокам с более низкими приоритетами. Приоритеты подразделяются на классы; обычным приложениям назначается приоритет класса "нормальный" (которому в характеристиках потока соответствует константа NORMAL_PRIORITY_CLASS). Некоторые потоки в системе могут иметь приоритеты реального времени (константа REAL_PRIORITY_CLASS); эти потоки также конкурируют друг с другом за процессорное время, однако до своего завершения не дают выполняться потокам с обычными приоритетами. Наконец, любой поток в системе, независимо от его приоритета, может быть вытеснен любым аппаратным и некоторыми программными прерываниями.

Программы ядра Windows NT, выполняемые на нулевом уровне привилегий, в частности драйверы NT, должны правильным образом взаимодействовать с аппаратурой компьютера. Поэтому им назначается специальный вид приоритета, называемый уровнем запроса прерывания (interrupt request level, IRQL). IRQL определяет приоритет потока по отношению к прерываниям от устройств компьютера. Обычно потоки ядра выполняются на нулевом уровне IRQL, которому соответствует символическое обозначение PASSIVE_LEVEL; некоторым программам, в частности процедурам отложенных прерываний, свойствен более высокий приоритет DISPATCH_LEVEL, равный двум. Уровни запросов прерываний устройств, обозначаемые DIRQL (от device interrupt request level), значительно выше и лежат в диапазоне от 12 (последний уровень ведомого контроллера) до 27 (первый уровень ведущего, к которому подключен системный таймер). Поэтому любые системные программы, кроме обработчиков прерываний, приостанавливают свое выполнение в случае прихода сигнала прерывания от любого устройства. Обработчикам аппаратных прерываний обычно назначаются приоритетные уровни IRQL, совпадающие с DIRQL соответствующего устройства. В результате реализуется система вложенных прерываний: обработчик прерываний от, скажем, мыши, работающий на уровне IRQL=23, прерывает свою работу в случае прихода прерывания от клавиатуры (уровень DIRQL=26) или, тем более, таймера (DIRQL=27), однако все аппаратные прерывания более низких уровней (от дисков и пр.) маскируются ядром Windows.

При установке в системе обработчика аппаратного прерывания, входящего в состав драйвера, ему необходимо назначить, во-первых, вектор прерывания, соответст-

вующий данному устройству, и, во-вторых, уровень запроса прерывания. Наша экспериментальная плата подключена к входу IRQ5 ведущего контроллера, которому в системах реального времени соответствует вектор 0xD=13. Однако в Windows NT этому входу контроллера назначается и другой уровень, и другой вектор. "Отображение" аппаратного вектора на систему векторов и уровней прерываний Windows осуществляется функцией HalGetInterruptVector(). Эта функция входит в состав компонента исполняющей системы ядра Windows, носящего название уровня аппаратных абстракций (Hardware Abstraction Layer, HAL) и отвечающего за организацию связи системы с аппаратурой компьютера. Функции HalGetInterruptVector() следует передать, в частности, тип используемой шины (ISA), ее номер (0), аппаратный уровень запроса прерывания (5 в нашем случае), а также адрес переменной, в которую будет записано значение DIRQL данного устройства. Перед вызовом функции HalGetInterruptVector() эта переменная (irql) должна содержать уровень прерывания, соответствующий реальному режиму (в нашем случае CARD_IRQ=5). После выполнения функции HalGetInterruptVector() в эту переменную будет помещено новое значение уровня, конкретно – 0x16. Функция возвращает значение отображенного вектора (0x35 для использованного уровня прерываний), который в нашей программе сохраняется на время в переменной MappedVector.

Получив отображенный вектор, можно зарегистрировать наш обработчик прерываний. Для этого вызывается функция IoConnectVector(), которой указываются в качестве параметров адрес обработчика (IsrRoutine), значения отображенного вектора (MappedVector) и отображенного уровня прерываний (irql) и другие данные. Функция IoConnectVector() возвращает через первый параметр адрес созданного ею объекта прерывания. Особо следует остановиться на третьем параметре этой функции, который в документации определяется как

```
IN PVOID ServiceContext
```

и представляет собой указатель на переменную неопределенного заранее типа. Эта переменная (точнее, область памяти) будет передана в процедуру обработки прерываний драйвера при ее активизации аппаратным прерыванием. Тем самым создается возможность обмена данными с обработчиком прерываний, который работает, как говорят, в другом контексте, нежели остальные процедуры драйвера. Обычно в качестве передаваемого указателя используют адрес расширения устройства или (как в нашем примере) адрес объекта устройства. Через этот указатель процедура обработки прерываний сможет получить доступ ко всему объекту устройства, а через него и к расширению устройства, если это понадобится.

Если ограничиться описанными выше операциями, то в системе не будет уровня отложенных прерываний и всю обработку прерываний придется возложить на программу обработчика прерываний. Такой подход допустим в тех случаях, когда обработка каждого прерывания проста и заключается, например, в чтении данных из портов устройства или посылке в устройство управляющего сигнала. Если же обработка прерывания сопряжена со значительным расходом процессорного времени, ее необходимо перенести на уровень отложенных прерываний. Как уже отмечалось выше, обработчик прерываний драйвера работает на весьма высоком уровне аппаратного прерывания устройства, когда выполнение практически всех остальных прикладных и системных программ запрещено. Тем самым на время обработки прерывания мы фактически выводим систему Windows из строя. Перевод обработки на уровень отло-

женных прерываний, для которого характерен низкий уровень приоритета (хотя и более высокий, чем для обычных программ), позволяет избежать нарушений в работе системы и в то же время обслужить наше устройство с достаточной эффективностью.

В рассматриваемом примере обслуживание каждого прерывания не занимает много времени и его можно было выполнить на уровне прерываний, не прибегая к услугам уровня отложенной обработки. Однако для общности в драйвере реализованы оба уровня. Инициализирующие действия по созданию уровня отложенных прерываний необходимо выполнить в той же процедуре инициализации драйвера DriverEntry(), для чего вызывается функция IoInitializeDpcRequest(). Эта функция регистрирует указанную ей функцию драйвера (DpcRoutine в нашем примере) в качестве процедуры отложенного вызова (defferred procedure call, DPC). Функция DpcRoutine() только регистрируется; активизировать ее нужно будет в обработчике прерываний.

Синхронизация обработчиков прерываний драйвера и приложения осуществляется с помощью события (см. статью 87). Для создания события прежде всего с помощью функции RtlInitUnicodeString() образуется его имя в формате структурной переменной со счетчиком типа UNICODE_STRING, а затем вызовом IoCreateNotificationEvent() создается само событие. Эта функция требует в качестве параметра имя события, а возвращает две характеристики события – дескриптор события через второй параметр и указатель на объект события, который мы сохраняем в расширении устройства под именем pEventObject.

Функция IoCreateNotificationEvent() назначает созданному событию установленное (signaled) состояние, что разрешает выполнение связанного с этим событием потока. Таким потоком у нас является обработчик прерываний приложения, который, естественно, не должен выполняться до прихода сигнала прерывания. Следовательно, событие необходимо перевести в сброшенное (nonsignaled) состояние и запретить тем самым работу обработчика приложения. Сброс события осуществляется вызовом функции ядра KeClearEvent(), которой передается в качестве параметра указатель на объект события. На этом функция инициализации драйвера завершается.

Прежде чем перейти к описанию обработчика прерываний драйвера, следует остановиться на роли кода действия IOCTL_START. Этот код действия приложение посылает в драйвер в последнем запросе ввода-вывода после того, как закончилась инициализация устройства засылкой соответствующих констант в регистры таймера и включением счета событий. Получив код действия IOCTL_START, драйвер сохраняет адрес текущего пакета запроса ввода-вывода в расширении устройства, устанавливает значения в блоке состояния IO_STATUS_BLOCK и вызовом функции IoMarkIrpPending() помечает текущий пакет IRP незавершенным, тем самым передавая его для завершения другим процедурам драйвера. Как видно из текста фрагмента драйвера, соответствующего коду действия IOCTL_START, этот фрагмент заканчивается не обычным для других фрагментов функции диспетчеризации вызовом функции IoCompleteRequest(), которая, считая, что пакет ввода-вывода обработан, возвращает его диспетчеру ввода-вывода, а просто оператором return. Пакет же ввода-вывода остается "висеть" в системе, что, между прочим, требует его обязательного завершения в той или иной процедуре драйвера. У нас завершение этого пакета будет выполнено в процедуре отложенной обработки прерываний.

Процедура обработки прерываний IsrRoutine() активизируется аппаратным прерыванием от счетчика-таймера и выполняется на уровне DIRQL. Как уже говорилось

выше, при ее активизации через второй параметр передается адрес той области памяти, которую мы "заказали" при регистрации этой процедуры. Мы в качестве этой области памяти использовали объект устройства; теперь мы извлекаем его в программу и через него получаем указатель на расширение устройства. Вспомним, что ранее мы поместили в расширение устройства адрес объекта созданного нами события, а также указатель на тот самый пакет ввода-вывода IRP, который мы пометили отложенным и который еще предстоит завершить. Последней операцией в процедуре IsrRoutine() является вызов функции диспетчера ввода-вывода IoRequestDpc(), ставящей в очередь на выполнение процедуру DpcRoutine() отложенной обработки прерываний. В качестве параметров этого вызова выступают объект устройства и текущий (а точнее, отложенный) IRP для этого устройства. Через последний параметр можно было передать в процедуру отложенной обработки дополнительную информацию.

Обратите внимание на форматы процедур немедленной и отложенной обработки прерываний. Их форматы жестко заданы Windows, поскольку эти процедуры вызываются системой асинхронным образом. В качестве второго и третьего параметров функции DpcRoutine() должны выступать указатели на объект устройства и пакет запроса ввода-вывода. Однако эти указатели не поставляются системой – это как раз те самые данные, которые мы передаем системной функции IoRequestDpc() при ее вызове. В то же время во входных параметрах процедуры обработки прерываний IsrRoutine() нет ни объекта устройства, ни IRP. Таким образом, забота о "добывании" этих параметров лежит на программисте. Мы и выполнили эту работу, сначала заказав передачу объекта устройства при вызове функции IoConnectInterrupt(), а затем сохранив указатель pIrp в расширении устройства.

Заметьте, что процедура обработки прерываний в случае ее успешного завершения должна возвращать значение TRUE.

Перейдем к рассмотрению последнего важного компонента драйвера – процедуры отложенной обработки прерываний. Задачей этой процедуры в нашем случае является активизация обработчика прерываний приложения, который пока находится в "спящем" состоянии ввиду того, что событие, управляющее потоком этого обработчика, сброшено.

Получив через поступившей в процедуру указатель на объект устройства адрес расширения, мы вызовом функции ядра KeSetEvent() устанавливаем наше событие во "взведенное" (signaled) состояние. В качестве параметра функции KeSetEvent() должен выступать адрес объекта события, а мы предусмотрительно сохранили его в расширении, откуда сейчас и извлекаем. Второй параметр функции KeSetEvent() позволяет изменить приоритет потока, а третий связан с организацией взаимодействующих потоков, что для нас не представляет интереса.

Установка события немедленно приводит к разблокировке ранее заблокированного событием потока. Выполнив разблокировку, следует вернуть событие в прежнее, сброшенное состояние, что и выполняется вызовом функции KeClearEvent(). После этого в процедуре устанавливаются значения блока состояния и текущий (отложенный) пакет ввода-вывода завершается вызовом функции IoCompleteRequest().

Приведенный ниже текст приложения (пример 91.1, продолжение) не содержит почти никаких новшеств. При выборе оператором пункта меню "Пуск таймера" с идентификатором MI_START вызывается прикладная функция StartTimer(), в которой последовательными запросами ввода-вывода осуществляется инициализация платы. В конце процедуры инициализации выполняется посылка в драйвер запроса ввода-

вывода с кодом действия IOCTL_START. Как было описано выше, этот запрос помечает текущий пакет ввода-вывода как задержанный, что обеспечивает правильную работу обеих процедур обработки прерываний, входящих в состав драйвера.

Далее функцией OpenEvent() открывается событие (которому мы дали имя "SignalEvent"), а функцией CreateThread() создается поток, состоящий из процедуры Isr(). Поскольку предпоследний параметр этой функции равен 0, поток немедленно запускается. Однако функция Isr() начинается с вызова WaitForSingleObject() с указанием имени нашего события, который блокирует дальнейшее выполнение потока до установки события. Блокировка осуществляется системными средствами и практически не потребляет процессорного времени. Установка события "SignalEvent" в процедуре драйвера DpcRoutine() снимает блокировку функции Isr(), которая с помощью двух запросов ввода-вывода читает содержимое выходного счетчика экспериментальной платы и выводит это число в окно сообщения с сопроводительной надписью. Вывод программы в конкретном прогоне приведен на рис. 91.2.

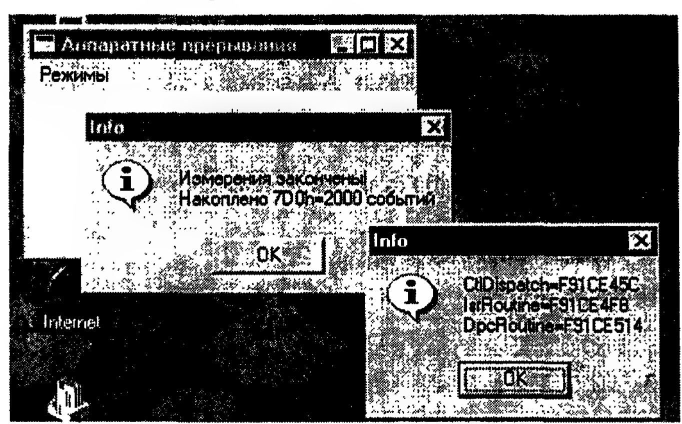


Рис. 91.2. Вывод приложения 91-01 для указанных в тексте программы значений констант

Пример 91.1 (продолжение). Тексты файлов приложения

Файл 91-01.H

```
#define MI_ADDR 100
#define MI_START 101
#define MI_DATA 102
#define MI_EXIT 104
#define IOCTL_ADDR (0x800<<2)|(0x22<<16)
#define IOCTL_READ (0x801<<2)|(0x22<<16)
#define IOCTL_WRITE (0x802<<2)|(0x22<<16)
#define IOCTL_START (0x803<<2)|(0x22<<16)
struct DATA{
USHORT Port;
UCHAR Value;
};
void Register(HINSTANCE);
void Create(HINSTANCE);
LRESULT CALLBACK WndProc(HWND,UINT,WPARAM,LPARAM);
BOOL OnCreate(HWND,LPCREATESTRUCT);
void OnCommand(HWND,int,HWND,UINT);
void OnDestroy(HWND);
void StartTimer();
void GetAddr();
```

```
USHORT GetData();
DWORD WINAPI Isr(LPVOID);
```

Файл 91-01.RC

```
#include "91-01.h"
Main MENU{
  POPUP "Режимы"  {
    MENUITEM "Адреса из драйвера",MI_ADDR
    MENUITEM "Пуск таймера",MI_START
    MENUITEM SEPARATOR
    MENUITEM "Выход",MI_EXIT
  }
}
```

Файл 91-01.CPP

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include "91-01.h"
char szClassName[]="MainWin";
char szTitle[]="Аппаратные прерывания";
HANDLE hDrv,hEvent;
DWORD cbRet,dwThreadID;
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE,LPSTR,int){
  MSG msg;
  Register(hInstance);
  Create(hInstance);
  while(GetMessage(&msg,NULL,0,0))DispatchMessage(&msg);
  return 0;
  }
void Register(HINSTANCE hInst){
  WNDCLASS wc;
  memset(&wc,0,sizeof(wc));
  wc.lpszClassName=szClassName;
  wc.hInstance=hInst;
  wc.lpfnWndProc=WndProc;
  wc.lpszMenuName="Main";
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
  wc.hbrBackground=GetStockBrush(WHITE_BRUSH);
  RegisterClass(&wc);
  }
void Create(HINSTANCE hInst){
  HWND hwnd=CreateWindow(szClassName,szTitle,WS_OVERLAPPEDWINDOW,
          10,10,250,150,HWND_DESKTOP,NULL,hInst,NULL);
  ShowWindow(hwnd,SW_SHOWNORMAL);
  }
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam){
  switch(msg)  {
    HANDLE_MSG(hwnd,WM_CREATE,OnCreate);
    HANDLE_MSG(hwnd,WM_COMMAND,OnCommand);
    HANDLE_MSG(hwnd,WM_DESTROY,OnDestroy);
    default:
    return(DefWindowProc(hwnd,msg,wParam,lParam));
    }
  }
BOOL OnCreate(HWND,LPCREATESTRUCT){
  hDrv = CreateFile("\\\\.\\Win32Name",GENERIC_READ|GENERIC_WRITE,0,
                     NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
  return TRUE;
  }
void OnCommand(HWND hwnd,int id,HWND,UINT){
```

```
  switch(id)  {
    case MI_ADDR:
      GetAddr();
      break;
    case MI_START:
      StartTimer();
      break;
    case MI_EXIT:
      DestroyWindow(hwnd);
    }
  }
void OnDestroy(HWND){
  PostQuitMessage(0);
  }
void StartTimer(){
DATA Data;
WORD Constant;
//Инициализируем  таймер
    Data.Port=0x30C;
    DeviceIoControl(hDrv,IOCTL_READ,&Data,3,&Data.Value,1,&cbRet,NULL);
    Data.Port=0x303;
    Data.Value=0x36;
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Value=0x70;
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Value=0xb6;
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
//Канал 0
    Constant=40;
    Data.Port=0x300;
    Data.Value=LOBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Value=HIBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
//Канал 1
    Constant=50000;
    Data.Port=0x301;
    Data.Value=LOBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Value=HIBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
//Инициализируем внутренний генератор
    Constant=1000;
    Data.Port=0x302;
    Data.Value=LOBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Value=HIBYTE(Constant);
    DeviceIoControl(hDrv,IOCTL_WRITE,&Data,3,NULL,0,&cbRet,NULL);
    Data.Port=0x30B;//Включаем счет
    DeviceIoControl(hDrv,IOCTL_READ,&Data.Port,2,&Data.Value,1,&cbRet,NULL);
//Посылаем запрос, сохраняющий задержанный (pending) IRP
    DeviceIoControl(hDrv,IOCTL_START,NULL,0,NULL,0,&cbRet,NULL);
    hEvent=OpenEvent(SYNCHRONIZE,FALSE,"SignalEvent");
    CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Isr,NULL,0,&dwThreadID);
    }
void GetAddr(){
    ULONG DrvAddr[3];
    char szText[80];
    DeviceIoControl(hDrv,IOCTL_ADDR,NULL,0,DrvAddr,12,&cbRet,NULL);
    wsprintf(szText,
        "CtlDispatch=%lX\nIsrRoutine=%lX\nDpcRoutine=%lX",
```

```
        DrvAddr[0],DrvAddr[1],DrvAddr[2]);
    MessageBox(NULL,szText,"Info",MB_ICONINFORMATION);
    }
//Обработчик прерывания приложения
DWORD WINAPI Isr(LPVOID){
  char szText[80];
  WaitForSingleObject(hEvent,INFINITE);
  ResetEvent(hEvent);
  USHORT Result=GetData();
  wsprintf(szText,
        "Измерения закончены!\nНакоплено %Xh=%d событий",Result,Result);
  MessageBox(NULL,szText,"Info",MB_ICONINFORMATION);
  return 0;
  }
//Функция чтения данных
USHORT GetData(){
    DATA Data;
    USHORT Result;
    Data.Port=0x309;//Старший байт результата
    DeviceIoControl(hDrv,IOCTL_READ,&Data.Port,2,&Data.Value,1,&cbRet,NULL);
    Result=(WORD)Data.Value;
    Result<<=8;
    Data.Port=0x308;//Младший байт результата
    DeviceIoControl(hDrv,IOCTL_READ,&Data.Port,2,&Data.Value,1,&cbRet,NULL);
    Result|=(WORD)Data.Value;
    return Result;
    }
```

Команды процессора

Ниже приводится алфавитный перечень команд процессоров Intel с кратким описанием действия каждой команды и примерами ее использования.

В *разделах статей*, начинающихся с обозначения **386+**, описываются отличия действия рассматриваемой команды в современных 32-разрядных процессорах (80386, i486, Pentium). Как правило, эти отличия заключаются в возможности использования не только 8- и 16-разрядных, но и 32-разрядных операндов, а также расширенных режимов адресации памяти. Обычные 16-разрядные программы реального режима вполне могут использовать расширенные регистры процессора (EAX, EBX и пр.), 32-битовые ячейки памяти и варианты команд для их обработки. Для того чтобы ассемблер правильно транслировал команды с 32-разрядными операндами, в программу необходимо включить одну из директив ассемблера .386, .486 или .586, а сегментам команд и данных придать описатель use16:

```
.386     ;Разрешение использовать средства процессора 80386
codes    segment use16 ;16-разрядный сегмент команд
assume   CS:codes
         ...
codes    ends
data     segment use16 ;16-разрядный сегмент данных
         ...
data     ends
```

Кроме этого, необходимо разрешить компоновщику обрабатывать 32-разрядные операнды, что для компоновщика TLINK осуществляется указанием ключа /3.

Отдельные статьи, начинающиеся с обозначений **386+** , **486+** и **Pentium+**, посвящены командам, отсутствующим в МП 86. Многие из этих команд (например, команды проверки бита bt или условной установки байта set) носят прикладной характер и могут использоваться в обычных программах реального режима.

Новые команды, реализованные впервые в МП 80386, сохраняют свое значение и в более современных процессорах. Для того, чтобы ассемблер распознавал команды МП 80386, в программе должна присутствовать директива .386.

Новые команды, реализованные впервые в МП 80486, сохраняют свое значение и в процессорах Pentium. Для того чтобы ассемблер распознавал команды МП 80486, в программе должна присутствовать директива .486.

Для того чтобы ассемблер распознавал команды, реализованные впервые в процессоре Pentium, в программе должна присутствовать директива .586.

Отдельные статьи, начинающиеся с обозначения **386P+**, посвящены привилегированным командам современных процессоров, работающих в защищенном режиме, и отсутствующим в МП 86. Для использования этих команд в программу необходимо включить одну из директив ассемблера .386P,.486P или .586P. Если при этом программа реализуется как 16-разрядное приложение MS-DOS, сегмент команд должен иметь описатель use16, так как при наличии директивы .386 транслятор по умолчанию создает 32-разрядное приложение.

AAA ASCII-коррекция регистра AX после сложения

Команда AAA используется вслед за операцией сложения в регистре AL двух распакованных двоично-десятичных чисел. Она преобразует результат сложения в неупакованное двоично-десятичное число, младший десятичный разряд которого находится в AL. Если результат превышает 9, выполняется инкремент содержимого регистра AH. Команда воздействует на флаги AF и CF.

AAD ASCII-коррекция регистра AX перед делением

Команда AAD используется перед операцией деления неупакованного двоично-десятичного числа в регистре AX на другое двоично-десятичное число. Команда преобразует делимое в регистре AX в беззнаковое двоичное число, чтобы в результате деления получились правильные неупакованные двоично-десятичные числа (частное – в AL, остаток – в AH). Команда воздействует на флаги SF, ZF и PF.

AAM ASCII-коррекция регистра AX после умножения

Команда AAM используется вслед за операцией умножения двух неупакованных двоично-десятичных чисел. Она преобразует результат умножения, являющийся двоичным числом, в правильное неупакованное двоично-десятичное число, младший разряд которого помещается в AL, а старший – в AH. Команда воздействует на флаги SF, ZF и PF.

AAS ASCII-коррекция регистра AL после вычитания

Команда AAS используется вслед за операцией вычитания одного неупакованного двоично-десятичного числа из другого в AL. Она преобразует результат вычитания в неупакованное двоично-десятичное число. Если результат вычитания оказывается меньше нуля, выполняется декремент содержимого регистра AH. Команда воздействует на флаги AF и CF; после ее выполнения AF=1, CF=1.

ADC Целочисленное сложение с переносом

Команда ADC осуществляет сложение первого и второго операндов, прибавляя к результату значение флага переноса CF. Исходное значение первого операнда (приемника) теряется, замещаясь результатом сложения. Второй операнд не изменяется. В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Команда воздействует на флаги OF, SF, ZF, AF, PF и CF.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров. Команда adc с 32-разрядными операндами может использоваться для сложения 64-разрядных целых чисел.

ADD Целочисленное сложение

Команда ADD осуществляет сложение первого и второго операндов. Исходное значение первого операнда (приемника) теряется, замещаясь результатом сложения. Второй операнд не изменяется. В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Команда воздействует на флаги OF, SF, ZF, AF, PF и CF.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

AND Логическое И

Команда AND осуществляет логическое (побитовое) умножение первого операнда на второй. Исходное значение первого операнда (приемника) теряется, замещаясь результатом умножения. В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами. Команда воздействует на флаги SF, ZF и PF.

Правила побитового умножения:

```
Первый операнд-бит  0 1 0 1
Второй операнд-бит  0 0 1 1
Бит результата      0 0 0 1
```

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

386P+ ARPL Коррекция запрашиваемого уровня привилегий селектора

Команда aprl сравнивает селектор с образцом, содержащим максимально допустимый уровень привилегий (обычно используется селектор CS) и устанавливает проверяемое значение в соответствии с меньшим из двух уровней привилегий. Если изменение уровня не потребовалось, флаг ZF сбрасывается, если потребовалось – устанавливается. В качестве первого операнда команды aprl может использоваться 16-разрядный регистр или слово памяти с проверяемым селектором; в качестве второго операнда – 16-разрядный регистр с селектором-образцом.

386+ BOUND Проверка индекса массива на выход за границы массива

Команда bound проверяет, лежит ли указанный индекс, рассматриваемый как число со знаком, внутри заданных вторым операндом границ. Если индекс выходит за границы массива снизу или сверху, генерируется прерывание с вектором 5. Первый операнд должен быть регистром, содержащим проверяемый индекс, второй – адресом поля памяти с двумя границами проверяемого массива. В команде bound допустимо использование как 16-битовых, так и 32-битовых операндов (но и первый и второй операнды должны быть одного типа).

386+ BSF Прямое сканирование битов

Команда bsf сканирует слово или двойное слово в поисках бита, равного единице. Сканирование выполняется от младшего бита (0) к старшему (15 или 31). Если в слове не найдено установленных битов, то устанавливается флаг ZF. Если установленные биты есть, то номер первого установленного бита заносится в указанный в команде регистр. Номером бита считается его позиция в слове, отсчитываемая от бита 0. В качестве первого операнда команды bsf следует указывать регистр, куда будет помещен результат сканирования, в качестве второго – регистр или ячейку памяти со сканируемым словом. В команде bsf допустимо использование как 16-битовых, так и 32-битовых операндов (но и первый и второй операнд должны быть одного типа).

386+ BSR Обратное сканирование битов

Команда bsf сканирует слово или двойное слово в поисках бита, равного единице. Сканирование выполняется от старшего бита (15 или 31) к младшему (0). Если в слове не найдено установленных битов, то устанавливается флаг ZF. Если установленные биты есть, то номер первого установленного бита заносится в указанный в команде ре-

гистр. Номером бита считается его позиция в слове, отсчитываемая от бита 0. В качестве первого операнда команды bsf следует указывать регистр, куда будет помещен результат сканирования, в качестве второго – регистр или ячейку памяти со сканируемым словом. В команде bsf допустимо использование как 16-битовых, так и 32-битовых операндов, но и первый и второй операнд должны быть одного типа (за исключением случая, когда второй операнд – константа).

486+ BSWAP Обмен байтов

Команда bswap изменяет порядок байтов в своем единственном операнде, в качестве которого может выступать только 32-разрядный регистр общего назначения. Биты 7...0 обмениваются с битами 31...24, а биты 15...18 – с битами 23...16. Другими словами, нумерация байтов регистра изменяется на противополжную (вместо 3, 2, 1, 0 – 0, 1, 2, 3). Команда не воздействует на флаги процессора.

386+ BT Проверка бита

Команда bt позволяет определить, установлен ли конкретный бит операнда. Анализируется первый операнд, номер бита задается вторым операндом. Первым операндом команды bt может служить регистр, ячейка или поле памяти (строка битов), вторым – регистр или непосредственное значение. В команде допустимо использование как 16-битовых, так и 32-битовых операндов, но и первый и второй операнд должны быть одной длины (за исключением случая, когда второй операнд – константа).

Значение проверяемого бита (0 или 1) копируется в флаг CF.

Номер проверяемого бита во втором операнде рассматривается как число со знаком. Таким образом, если в качестве первого операнда выступает строка битов, то проверяемый бит может располагаться относительно указанного адреса строки в диапазоне от –32 Кбит до 32 Кбит – 1 для 16-разрядных операндов или от –4 Гбит до 4 Гбит – 1 для 32-разрядных операндов.

Если в качестве второго операнда используется непосредственное значение, то допускается анализ поля памяти размером не более размера второго операнда (слова или двойного слова). При необходимости анализировать более длинные строки битов номер проверяемого бита должен содержаться в регистре.

386+ BTC Проверка и инверсия бита

Команда btc проверяет определенный бит первого операнда, копирует его значение в флаг CF и после этого инвертирует. Номер бита задается вторым операндом. Первым операндом команды btc может служить регистр, ячейка или поле памяти (строка битов), вторым – регистр или непосредственное значение. В команде допустимо использование как 16-битовых, так и 32-битовых операндов, но и первый и второй операнд должны быть одной длины (за исключением случая, когда второй операнд – константа).

Номер проверяемого бита во втором операнде рассматривается как число со знаком. Таким образом, если в качестве первого операнда выступает строка битов, то проверяемый бит может располагаться относительно указанного адреса строки в диапазоне от –32 Кбит до 32 Кбит – 1 для 16-разрядных операндов или от –4 Гбит до 4 Гбит – 1 для 32-разрядных операндов.

Если в качестве второго операнда используется непосредственное значение, то допускается анализ поля памяти размером не более размера второго операнда (слова или двойного слова). При необходимости анализировать более длинные строки битов номер проверяемого бита должен содержаться в регистре.

386+ BTR Проверка и сброс бита

Команда btr проверяет определенный бит первого операнда, копирует его значение в флаг CF и после этого сбрасывает. Номер бита задается вторым операндом. Первым операндом команды btr может служить регистр, ячейка или поле памяти (строка битов), вторым – регистр или непосредственное значение. В команде допустимо использование как 16-битовых, так и 32-битовых операндов, но и первый и второй операнд должны быть одной длины (за исключением случая, когда второй операнд – константа).

Номер проверяемого бита во втором операнде рассматривается как число со знаком. Таким образом, если в качестве первого операнда выступает строка битов, то проверяемый бит может располагаться относительно указанного адреса строки в диапазоне от –32 Кбит до 32 Кбит – 1 для 16-разрядных операндов или от –4 Гбит до 4 Гбит – 1 для 32-разрядных операндов.

Если в качестве второго операнда используется непосредственное значение, то допускается анализ поля памяти размером не более размера второго операнда (слова или двойного слова). При необходимости анализировать более длинные строки битов номер проверяемого бита должен содержаться в регистре.

386+ BTS Проверка и установка бита

Команда bts проверяет определенный бит первого операнда, копирует его значение в флаг CF и после этого сбрасывает. Номер бита задается вторым операндом. Первым операндом команды bts может служить регистр, ячейка или поле памяти (строка битов), вторым – регистр или непосредственное значение. В команде допустимо использование как 16-битовых, так и 32-битовых операндов, но и первый и второй операнд должны быть одной длины (за исключением случая, когда второй операнд – константа).

Номер проверяемого бита во втором операнде рассматривается как число со знаком. Таким образом, если в качестве первого операнда выступает строка битов, то проверяемый бит может располагаться относительно указанного адреса строки в диапазоне от –32 Кбит до 32 Кбит – 1 для 16-разрядных операндов или от –4 Гбит до 4 Гбит – 1 для 32-разрядных операндов.

Если в качестве второго операнда используется непосредственное значение, то допускается анализ поля памяти размером не более размера второго операнда (слова или двойного слова). При необходимости анализировать более длинные строки битов номер проверяемого бита должен содержаться в регистре.

CALL Вызов процедуры

Команда CALL передает управление процедуре (подпрограмме), сохранив перед этим в стеке адрес возврата. Команда RET, которой обычно заканчивается процедура, забирает из стека адрес возврата и возвращает управление на команду, следующую за командой CALL. Команда CALL имеет 4 модификации:

- вызов прямой ближний (в пределах текущего программного сегмента);
- вызов прямой дальний (вызов процедуры, расположенной в другом программном сегменте);
- вызов косвенный ближний;
- вызов косвенный дальний.

Команда CALL прямого ближнего вызова заносит в стек смещение точки возврата в текущем программном сегменте и модифицирует IP так, чтобы в нем содержалось

смещение точки перехода в том же программном сегменте. Необходимое для вычисления этого смещения расстояние до точки перехода содержится в коде команды, который занимает 3 байта (код операции E8h и смещение к точке перехода).

Команда CALL прямого дальнего вызова заносит в стек два слова – сначала сегментный адрес текущего программного сегмента, а затем (выше, в слово с меньшим адресом) смещение точки возврата в текущем программном сегменте. Далее модифицируются регистры IP и CS: в IP помещается смещение точки перехода в том сегменте, куда осуществляется переход, а в CS – сегментный адрес этого сегмента. Обе эти величины берутся из кода команды, который занимает 5 байт (код операции 9Ah, относительный адрес вызываемой процедуры и ее сегментный адрес).

Косвенные вызовы отличаются тем, что адрес перехода извлекается не из кода команды, а из ячеек памяти или регистров; в коде команды содержится информация о том, где находится адрес перехода. Поэтому длина кода команды зависит от используемого способа адресации.

CBW Преобразование байта в слово

Команда CBW заполняет регистр AH знаковым битом числа, находящегося в регистре AL, что дает возможность выполнять арифметические операции над исходным операндом-байтом как над словом в регистре AX. Команда не имеет параметров и не воздействует на флаги процессора.

386+ CDQ Преобразование двойного слова в четверное

Команда cdq расширяет знак двойного слова в регистре EAX на регистр EDX. Эту команду можно использовать для образования четырехсловного делимого из двухсловного перед операцией двухсловного деления. Команда не имеет параметров и не воздействует на флаги процессора.

CLC Сброс флага переноса

Команда CLC сбрасывает флаг переноса CF в регистре флагов. Команда не имеет параметров и не воздействует на остальные флаги процессора.

CLD Сброс флага направления

Команда CLD сбрасывает флаг DF в регистре флагов, устанавливая прямое (в порядке возрастания адресов) направление выполнения операций со строками. Команда не имеет параметров и не воздействует на остальные флаги процессора.

CLI Сброс флага прерываний

Команда CLI сбрасывает флаг IF в регистре флагов, запрещая все аппаратные прерывания. Прерывания будут оставаться запрещенными до установки флага IF командой sti. На программные прерывания (команду int) флаг не действует. Команда не имеет параметров и не воздействует на остальные флаги процессора.

386P+ CLTS Сброс флага переключения задачи в управляющем регистре 0

Команда clts сбрасывает флаг TS в регистре CR0.

CMC Инвертирование флага переноса

Команда CMC изменяет значение флага CF в регистре флагов на обратное. Команда не имеет параметров и не воздействует на остальные флаги процессора.

CMP Сравнение

Команда CMP выполняет вычитание второго операнда из первого. В соответствии с результатом вычитания устанавливаются состояния флагов CF, PF, AF, ZF, SF и OF.

Сами операнды не изменяются. Таким образом, если команду сравнения записать в общем виде

```
cmp     операнд_1, операнд_2
```

то ее действие можно условно изобразить следующим образом:

```
операнд_1 - операнд_2 → флаги процессора
```

В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Обычно вслед за командой CMP стоит одна из команд условных переходов, анализирующих состояние флагов процессора (je – переход, если равно; jne – переход, если не равно, и т. д.).

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

CMPS Сравнение строк
CMPSB Сравнение строк по байтам
CMPSW Сравнение строк по словам

Команды предназначены для операций над строками (строкой называется последовательность байтов или слов с любым содержимым). Они сравнивают по одному элементу каждой строки, осуществляя вычитание второго операнда из первого и устанавливая в соответствии с результатом вычитания флаги CF, PF, AF, ZF, SF и OF. Первый операнд адресуется через DS:SI, второй – через ES:DI. Операцию сравнения можно условно изобразить следующим образом:

```
(DS:SI) - (ES:DI) → флаги процессора
```

После каждой операции сравнения регистры SI и DI получают положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера сравниваемых элементов.

Вариант команды CMPS имеет формат

```
cmps    строка_1, строка_2
```

(что не избавляет от необходимости инициализировать регистры DS:SI и ES:DI адресами строк строка_1 и строка_2 соответственно). В этом формате возможна замена сегмента первой строки:

```
cmps    ES:строка_1, строка_2
```

Рассматриваемые команды могут предваряться префиксами повторения REPE/REPZ (повторять, пока элементы равны, т. е. до первого неравенства) и REPNE/REPNZ (повторять, пока элементы не равны, т. е. до первого равенства). В любом случае выполняется не более CX операций над последовательными элементами.

После выполнения рассматриваемых команд регистры SI и DI указывают на ячейки памяти, находящиеся за теми (если DF=0) или перед теми (если DF=1) элементами строк, на которых закончились операции сравнения.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

386+ CMPSD Сравнение строк по двойным словам

Команда аналогична командам МП 86 cmpsb и cmpsw, но позволяет сравнивать 32-битовые участки строк, адресуемых через регистры DS:ESI и ES:EDI (в 16-

разрядных приложениях – через DS:SI и ES:DI). Использование мнемоники cmpsd с префиксом rep не означает, что в качестве счетчика будет автоматически использоваться расширенный регистр ECX.

486+ CMPXCHG Сравнение и обмен

Команда cmpxchg выполняет в одной операции сравнение и обмен операндов. Команда требует два параметра и неявным образом использует третий операнд – регистр EAX. Первый операнд (приемник) должен находиться в 16- или 32-битовой ячейке памяти, второй операнд (источник) – в регистре общего назначения такого же размера. Команда выполняет сравнение операнда-приемника с содержимым неявного операнда – регистра EAX. Если сравниваемые значения совпадают, операнд-приемник замещается операндом-источником (т. е. содержимое регистра записывается в память). Если сравниваемые значения не совпадают, содержимое памяти (приемник) поступает в регистр EAX (рис. П-1.1). Команда воздействует на флаги OF, SF, ZF, AF, PF и CF.

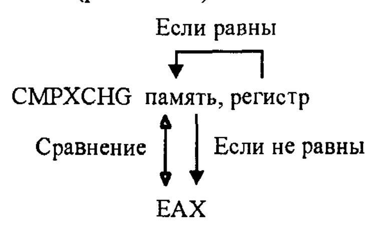


Рис. П-1.1. Действие команды cmpxchg

Pentium+ CMPXCHG8B Сравнение и обмен 8 байт

Команда cmpxchg8b выполняет в одной операции сравнение и обмен 8-байтовых операндов. Команда требует один параметр и неявным образом использует еще два операнда – пары регистров EDX:EAX и ECX:EBX. В качестве явного операнда команды (приемника) может выступать только 64-битовая (8-байтовая) ячейка памяти. Команда выполняет сравнение операнда-приемника в памяти с содержимым EDX:EAX. Если сравниваемые значения совпадают, то операнд-приемник в памяти замещается 64-битным значением ECX:EBX. Если сравниваемые значения не совпадают, содержимое памяти поступает в пару регистров EDX:EAX, замещая один из сравниваемых операндов (рис. П-1.2). Команда воздействует на флаг ZF.

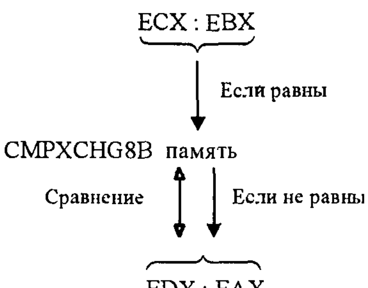


Рис. П-1.2. Действие команды cmpxchg8b

Pentium+ CPUID Идентификация процессора

Команда cpuid позволяет получить код идентификации процессора, установленного на данном компьютере. Команда в качестве неявного операнда использует регистр EAX. Для процессоров Pentium регистр EAX перед вызовом команды cpuid может принимать два значения: 0 и 1. Если EAX=0, то команда возвращает в регистре EAX код 1, а в регистрах EBX, EDX и ECX (именно в таком порядке) – три части символь-

ной строки, идентифицирующей изготовителя процессора. Для процессоров Intel возвращаемая строка в целом имеет вид "GenuineIntel".

Если перед вызовом команды cpuid значение EAX равно единице, то команда возвращает в регистре EAX коды разработки конкретной версии процессора, а в регистре EDX-код 1BFh, содержащий информацию о возможностях процессора.

Коды разработки в регистре EAX хранятся в следующем формате:

- биты 0...3 – номер поколения (например, 3);
- биты 4...7 – модель (например, 4);
- биты 8...11 – семейство (5 для Pentium).

Содержимое регистра EDX включает конфиденциальную информацию изготовителя, а также говорит о наличии на кристалле микропроцессора арифметического сопроцессора (бит 0) и поддержке команды cmpxchg8b (бит 8).

CWD Преобразование слова в двойное слово

Команда CWD заполняет регистр DX знаковым битом содержимого регистра AX, преобразуя тем самым 16-разрядное число со знаком в 32-разрядное. Команду удобно использовать для преобразования 2-байтового делимого в 4-байтовое (двойное слово) при делении на 16-разрядный операнд. Команда не имеет параметров и не воздействует на флаги процессора.

386+ CWDE Преобразование слова в двойное слово с расширением

Команда cwde заполняет старшую половину регистра EAX знаковым битом содержимого регистра AX, преобразуя тем самым 16-разрядное число со знаком в 32-разрядное, размещаемое в расширенном регистре EAX. Команда не имеет операндов и не воздействует на флаги процессора.

DAA Десятичная коррекция в регистре AL после сложения

Команда DAA корректирует результат сложения в регистре AL двух упакованных десятичных чисел (по одной цифре в каждом полубайте), чтобы получить пару правильных упакованных десятичных цифр. Команда используется вслед за операцией сложения упакованных десятичных чисел. Если результат сложения превышает 99, возникает перенос и устанавливается флаг CF. Команда воздействует на флаги SF, ZF, AF, PF и CF.

DAS Десятичная коррекция в регистре AL после вычитания

Команда DAS корректирует результат вычитания в регистре AL двух упакованных десятичных чисел (по одной цифре в каждом полубайте), чтобы получить пару правильных упакованных десятичных цифр. Команда используется вслед за операцией вычитания упакованных десятичных чисел. Если для вычитания требовался заем, устанавливается флаг CF. Команда воздействует на флаги SF, ZF, AF, PF и CF.

DEC Декремент (уменьшение на 1)

Команда DEC вычитает 1 из операнда, в качестве которого можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение. Операнд интерпретируется как число без знака. Команда воздействует на флаги OF, SF, ZF, AF и PF.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

DIV Деление целых беззнаковых чисел

Команда DIV выполняет деление целого числа без знака, находящегося в регистрах AX (в случае деления на байт) или DX:AX (в случае деления на слово), на операнд-

источник (целое число без знака). Размер делимого в два раза больше размеров делителя и остатка.

Для 1-байтовых операций делимое помещается в регистр AX; после выполнения операции частное записывается в регистр AL, а остаток – в регистр AH.

Для 2-байтовых операций делимое помещается в регистры DX:AX (в DX – старшая часть, в AX – младшая); после выполнения операции частное записывается в регистр AX, а остаток – в регистр DX.

В качестве операнда-делителя можно указывать регистр данных или ячейку памяти; не допускается деление на непосредственное значение. Если делитель равен нулю или если частное не помещается в назначенный регистр, возбуждается прерывание через вектор 0. Команда не воздействует на флаги процессора.

Команду div можно использовать для целочисленного деления неупакованного двоично-десятичного числа в регистре AX не неупакованный двоично-десятичный делитель, если перед ней выполнить команду aad.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров. При этом если делитель представляет 32-битовую величину, то возможен только один вариант команды деления, когда делимое находится в паре регистров EDX:EAX. В этом случае частное будет помещено в регистр EAX, остаток – в EDX.

386+ ENTER Создание стекового кадра для параметров процедуры

Команда enter, обычно используемая в качестве первой команды процедуры, выделяет заданный объем стекового пространства для локальных (автоматических) параметров процедуры, предоставляя процедуре указатель на выделенную область (в качестве такого указателя используется регистр EBP) и смещая указатель стека ESP так, чтобы он указывал на начало свободного стекового пространства. В результате процедура имеет возможность обращаться по ходу своего выполнения к своим локальным параметрам и в то же время пользоваться оставшимся пространством стека для временного сохранения в нем любых данных с помощью команд push и pop. Команда leave в конце процедуры выполняет обратные действия, возвращая стек в исходное состояние и уничтожая область локальных переменных. Локальными, как известно, называются как раз те переменные, которые существуют только в течение времени выполнения некоторой процедуры и автоматически исчезают после ее завершения.

Команды enter и leave включаются в текст программы многими трансляторами языков высокого уровня для управления доступом к локальным переменным вложенных процедур.

HLT Останов

Команда hlt прекращает выполнение программы и переводит процессор в состояние останова. Работа процессора возобновляется после операции запуска, а также в случае прихода немаскируемого или разрешенного маскируемого прерывания.

IDIV Деление целых знаковых чисел

Команда IDIV выполняет деление целого числа со знаком, находящегося в регистрах AX (в случае деления на байт) или DX:AX (в случае деления на слово), на операнд-источник (целое число со знаком). Размер делимого в два раза больше размеров делителя и остатка. Оба результата рассматриваются как числа со знаком, причем знак остатка равен знаку делимого.

Для 1-байтовых операций делимое помещается в регистр AX; после выполнения операции частное записывается в регистр AL, а остаток – в регистр AH.

Для 2-байтовых операций делимое помещается в регистры DX:AX (в DX – старшая часть, в AX – младшая); после выполнения операции частное записывается в регистр AX, а остаток – в регистр DX.

В качестве операнда-делителя можно указывать регистр данных или ячейку памяти; не допускается деление на непосредственное значение. Если делитель равен нулю или если частное не помещается в назначенный регистр, возбуждается прерывание через вектор 0. Команда не воздействует на флаги процессора.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров. При этом если делитель представляет 32-битовую величину, то возможен только один вариант команды деления, когда делимое находится в паре регистров EDX:EAX. В этом случае частное будет помещено в регистр EAX, остаток – в EDX.

IMUL Умножение целых знаковых чисел

Команда IMUL выполняет умножение целого знакового числа, находящегося в регистре AL (в случае деления на байт) или AX (в случае деления на слово), на операнд-источник (целое число со знаком). Размер произведения в два раза больше размера сомножителей.

Для 1-байтовых операций один из сомножителей помещается в регистр AL; после выполнения операции произведение записывается в регистр AX.

Для 2-байтовых операций один из сомножителей помещается в регистр AX; после выполнения операции произведение записывается в регистры DX:AX (в DX – старшая часть, в AX – младшая).

В качестве операнда-сомножителя можно указывать регистр данных или ячейку памяти; не допускается умножение на непосредственное значение. Команда воздействует на флаги OF и CF. Если AH или DX представляют собой просто знаковое расширение AL или AX соответственно (т. е. результат умножения со знаком верен), OF и CF сбрасываются в 0; в противном случае (результат со знаком не помещается в AX или DX:AX) OF и CF устанавливаются в 1.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров. Имеются также варианты команды с двумя и тремя операндами.

Для команды imul с одним операндом второй сомножитель должен располагаться в AL, AX или EAX. Процессор выбирает размерность второго сомножителя исходя из размерности первого, указанного в качестве операнда; 16-, 32- или 64-битовый знаковый результат помещается в регистры AX, DX:AX или EDX:EAX соответственно. Если после операции умножения содержимое AH, DX или EDX является лишь знаковым расширением AL, AX или EAX соответственно, то флаги CF и OF сбрасываются в 0. В противном случае они устанавливаются в 1.

Для команды imul с двумя операндами их произведение записывается в первый операнд; второй операнд не изменяется. В качестве первого операнда могут выступать 16- или 32-разрядные регистры общего назначения; в качестве второго операнда – 16- или 32-разрядные регистры общего назначения, 16- или 32-битовые ячейки памяти или непосредственное значение. Оба операнда должны иметь один размер. Если результат

умножения помещается в первый операнд, флаги CF и OF сбрасываются в 0. В противном случае они устанавливаются в 1.

Для команды imul с тремя операндами произведение второго и третьего операндов записывается в первый операнд. В качестве первого операнда могут выступать 16- или 32-разрядные регистры общего назначения; в качестве второго операнда – 16- или 32-разрядные регистры общего назначения или 16- или 32-битовые ячейки памяти; в качестве третьего операнда – только непосредственное значение. Два первых операнда должны иметь один размер. Если результат умножения помещается в первый операнд, флаги CF и OF сбрасываются в 0. В противном случае они устанавливаются в 1.

IN Ввод из порта

Команда IN вводит в регистры AL или AX соответственно байт или слово из порта, указываемого вторым операндом. Адрес порта помещается в регистр DX. Если адрес порта не превышает 255, он может быть указан непосредственным значением. Указание регистра-приемника (AL или AX) обязательно. Команда не воздействует на флаги процессора.

386+ Допустимо использование в качестве операнда-приемника расширенного регистра EAX (если адресуемое устройство позволяет прочитать из его порта двойное слово).

INC Инкремент (увеличение на 1)

Команда INC прибавляет 1 к операнду, в качестве которого можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение. Операнд интерпретируется как число без знака. Команда воздействует на флаги OF, SF, ZF, AF и PF. Команда не воздействует на флаг CF; если требуется воздействие на этот флаг, необходимо использовать команду add *op*,1.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

386+
INS Ввод строки из порта
INSB Ввод байта из порта
INSW Ввод слова из порта
INSD Ввод двойного слова из порта

Команды предназначены для ввода данных из порта непосредственно в память. Адрес порта указывается, как и для команды in, в регистре DX, при этом задание адреса порта непосредственным значением не допускается. Данные пересылаются по адресу, находящемуся в паре регистров ES:EDI. Замена сегмента не допускается. Команда insb переносит из порта 1 байт, команда insw – 1 слово, команда insd – 1 двойное слово, а команда ins может быть использована для передачи байтов, слов и двойных слов. В последнем случае размер загружаемого данного определяется описанием строки (с помощью директив db, dw или dd). После передачи данных регистр EDI получает положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1, 2 или 4, в зависимости от размера передаваемых данных.

Вариант команды ins (не insb, insw или insd) имеет формат

```
ins    строка, DX
```

(что не избавляет от необходимости инициализировать регистры ES:EDI адресом строки).

Если устройство, адресуемое через порт, может передавать последовательность данных, то команды ins можно предварить префиксом повторения rep. В этом случае из порта принимается CX элементов данных заданного размера.

Команды ins не воздействуют на флаги процессора.

INT Программное прерывание

Команда INT инициирует в процессоре процедуру прерывания, в результате которой управление передается на программу обработки прерывания с номером n, который указан в качестве операнда команды INT. В стек прерываемого процесса (текущей программы) заносится содержимое регистра флагов, сегментного регистра CS и указателя команд IP, после чего в регистры IP и CS передается содержимое двух слов из вектора прерывания типа n (расположенных по адресам 0:n*4 и 0:n*4+2). Команда INT сбрасывает флаг IF.

INTO Прерывание по переполнению

Команда INTO, будучи установлена вслед за какой-либо арифметической, логической или строковой командой, возбуждает процедуру прерывания типа 4, если предшествующая команда установила флаг переполнения OF. Перед использованием команды INTO пользователь должен поместить в вектор прерывания 4 двухсловный адрес своего обработчика прерывания по переполнению. Команда сбрасывает флаги IF и TF в 0. Команда iret, которой всегда завершается обработчик прерывания, восстанавливает исходное состояние этих флагов.

IRET Возврат из программы обработки прерывания

Команда IRET возвращает управление прерванному в результате аппаратного или программного прерывания процессу. Команда извлекает из стека три верхних слова и помещает их в регистры IP, CS и флагов (см. команду INT). Командой IRET должна завершаться любая программа обработки прерываний, как аппаратных, так и программных (от команды int). Команда не воздействует на флаги, однако она загружает в регистр флагов из стека его исходное содержимое, которое было там сохранено процессором в ходе обслуживания прерывания. Если требуется, чтобы после возврата из обработчика программного прерывания командой iret какие-либо флаги процессора были установлены требуемым образом (весьма распространенный прием), их установку надо выполнить в копии флагов в стеке.

386+ IRETD Возврат из прерывания в 32-разрядном режиме

Команда iretd используется в защищенном режиме для возврата из обработчика прерывания или исключения, а также для переключения на исходную задачу. В отличие от 16-разрядной команды iret, данная команда, завершая обработку прерывания или исключения, снимает со стека три двойных слова, содержащих расширенный регистр флагов EFALGS, CS и расширенный указатель команд EIP. В случае переключения задач команда iretd выполняет переключение контекстов задач – сохранение состояния завершающейся задачи в ее сегменте состояния задачи и загрузку регистров процессора из сегмента состояния исходной задачи.

Jcc Команды условных переходов

Команды условных переходов осуществляют переход по указанному адресу при выполнении условия, заданного мнемоникой команды. Если заданное условие не выполняется, переход не осуществляется, а выполняется команда, следующая за конкрет-

ной командой условного перехода. Переход может осуществляться как вперед, так и назад в диапазоне +127...–128 байт.

Команды условных переходов перечислены в табл. П-1.1.

386+ Команды условных переходов имеют варианты 16- и 32-разрядной адресации (при тех же мнемонических обозначениях) и могут передавать управление в диапазоне –32768...+32767 байт для сегментов с атрибутом размера 16 и в диапазоне $-2^{31}...+2^{31}$ –1 байт для сегментов с атрибутом размера 32.

Таблица П-1.1. Команды условных переходов и их действие

Команда	*Перейти,если*	*Условие перехода*
JA	выше	CF=0 и ZF=0
JAE	выше или равно	CF=0
JB	ниже	CF=1
JBE	ниже или равно	CF=1 или ZF=1
JC	перенос	CF=1
JCXZ	CX=0	CX=0
JE	равно	ZF=1
JG	больше	ZF=0 или SF=OF
JGE	больше или равно	SF=OF
JL	меньше	SF не равно OF
JLE	меньше или равно	ZF=1 или SF не равно OF
JNA	не выше	CF=1 или ZF=1
JNAE	не выше и не равно	CF=1
JNB	не ниже	CF=0
JNBE	не ниже и не равно	CF=0 и ZF=0
JNC	нет переноса	CF=0
JNE	не равно	ZF=0
JNG	не больше	ZF=1 или SF не равно OF
JNGE	не больше и не равно	SF не равно OF
JNL	не меньше	SF=OF
JNLE	не меньше и не равно	ZF=0 и SF=OF
JN	нет переполнения	OF=0
JNP	нет четности	PF=0
JNS	знаковый бит равен нулю	SF=0
JNZ	не нуль	ZF=0
JO	переполнение	OF=1
JP	есть четность	PF=1
JPE	сумма битов четная	PF=1
JPO	сумма битов нечетная	PF=0
JS	знаковый бит равен 1	SF=1
JZ	нуль	ZF=1

JMP Безусловный переход

Команда JMP передает управление в указанную точку того же или другого программного сегмента. Адрес возврата не сохраняется. Команда не воздействует на флаги процессора.

Команда JMP имеет 5 разновидностей:

- переход прямой короткий (в пределах –128...+127 байт);
- переход прямой ближний (в пределах текущего сегмента команд);

- переход прямой дальний (в другой сегмент команд);
- переход косвенный ближний;
- переход косвенный дальний.

Все разновидности переходов имеют одну и ту же мнемонику JMP, хотя и различающиеся коды операций. В некоторых случаях транслятор может определить вид перехода по контексту, в тех же случаях, когда это невозможно, следует использовать атрибутные операторы:

- short – прямой короткий переход;
- near ptr – прямой ближний переход;
- far ptr – прямой дальний переход;
- word ptr – косвенный ближний переход;
- dword ptr – косвенный дальний переход.

386+ Допустимо использование дополнительных режимов адресации 32-разрядных процессоров. Для 32-разрядных приложений допустимо использование 32-битовых операндов. В защищенном режиме вместо сегментного адреса сегмента (при дальних переходах) выступает его селектор.

LAHF Загрузка флагов в регистр AH

Команда LAHF копирует флаги SF, ZF, AF, PF и CF соответственно в разряды 7, 6, 4, 2 и 0 регистра AH. Значение битов 5, 3 и 1 не определено. Команда не имеет параметров и не изменяет флаги процессора.

Команда LAHF (совместно с командой SAHF) дает возможность читать и изменять значение флагов процессора, в том числе флагов SF, ZF, AF и PF, которые нельзя изменить непосредственно.

386P+ LAR Загрузка прав доступа

Команда lar загружает в первый операнд (16- или 32-разрядный регистр) поле атрибутов сегмента из дескриптора сегмента, заданного селектором во втором операнде. В качестве операнда с селектором может использоваться 16- или 32-разрядный регистр или ячейка памяти. В операнд-приемник поступают 2 байта атрибутов селектора с замаскированным полем старших битов границы сегмента.

LDS Загрузка указателя с использованием регистра DS

Команда LDS считывает из памяти по указанному адресу двойное слово, содержащее указатель (полный адрес некоторой ячейки), и загружает младшую половину указателя (т. е. относительный адрес) в указанный в команде регистр, а старшую половину указателя (т. е. сегментный адрес) – в регистр DS. Таким образом, команда

```
lds    reg,mem
```

эквивалентна по результату следующей группе команд:

```
mov    reg,word ptr mem
mov    DS,word ptr mem+2
```

В качестве первого операнда команды LDS должен быть указан регистр общего назначения, в качестве второго – ячейка памяти. Команда не воздействует на флаги процессора.

386+ Допустимо использование 32-разрядного регистра-приемника и 32-битового смещения в памяти, а также дополнительных режимов адресации 32-разрядных про-

цессоров. В защищенном режиме вместо сегментного адреса сегмента выступает его селектор.

LEA Загрузка исполнительного адреса

Команда LEA загружает в регистр, указанный в команде в качестве первого операнда, относительный адрес второго операнда. В качестве первого операнда следует указывать регистр общего назначения, в качестве второго – ячейку памяти. Команда

```
lea    reg,mem
```

по своему результату эквивалентна команде

```
mov    reg,offset mem
```

Команда не воздействует на флаги процессора.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

386+ LEAVE Выход из процедуры высокого уровня

Команда leave выполняет действия, противоположные действиям последней команды enter. Она логически уничтожает созданный командой enter стековый кадр со всеми содержащимися в нем локальными переменными и подготавливает стек к выполнению команды iret, завершающей переход в вызывающую процедуру. Команда leave не имеет параметров. Более подробное описание и пример см. в описании команды enter.

LES Загрузка указателя с использованием регистра ES

Команда LES считывает из памяти по указанному адресу двойное слово, содержащее указатель (полный адрес некоторой ячейки), и загружает младшую половину указателя (смещение) в указанный в команде регистр, а старшую половину указателя (сегментный адрес) – в регистр ES. Таким образом, команда

```
les    reg,mem
```

эквивалентна по результату следующей паре команд:

```
mov    reg,word ptr mem
mov    ES,word ptr mem+2
```

В качестве первого операнда команды должен быть указан регистр общего назначения, в качестве второго – ячейка памяти. Команда не воздействует на флаги процессора.

386+ Допустимо использование 32-разрядного регистра-приемника и 32-битового смещения в памяти, а также дополнительных режимов адресации 32-разрядных процессоров. В защищенном режиме вместо сегментного адреса сегмента выступает его селектор.

386+
LFS Загрузка указателя с использованием регистра FS
LGS Загрузка указателя с использованием регистра FS
LSS Загрузка указателя с использованием регистра FS

Команды считывают из памяти полный указатель, состоящий из селектора и 16- или 32-битового смещения, и загружают младшую половину указателя (т. е. относительный адрес) в указанный в команде регистр общего назначения, а старшую половину указателя (т. е. селектор) – в сегментный регистр, указанный в мнемонике команды.

В качестве первого операнда всех перечисленных команд указывается 16- или 32-разрядный регистр общего назначения; в качестве второго – ячейка памяти с 32- или 48-битовым содержимым. Команда не воздействует на флаги процессора.

Примеры см. в описании команд lds и les.

386P+ LGDT Загрузка регистра таблицы глобальных дескрипторов

Команда lgdt загружает регистр таблицы глобальных дескрипторов (GDTR) из 48-битового псевдодескриптора, содержащего 32-битовый базовый адрес и 16-битовую границу таблицы глобальных дескрипторов, находящейся в памяти. В качестве операнда команды lgdt выступает относительный адрес псевдодескриптора.

386P+ LIDT Загрузка регистра таблицы дескрипторов прерываний

Команда lidt загружает регистр таблицы дескрипторов прерываний (IDTR) из 48-битового псевдодескриптора, содержащего 32-битовый базовый адрес и 16-битовую границу таблицы дескрипторов прерываний, находящейся в памяти. В качестве операнда команды lidt выступает относительный адрес псевдодескриптора.

386P+ LLDT Загрузка регистра таблицы локальных дескрипторов

Команда lldt загружает регистр таблицы локальных дескрипторов (LDTR) селектором, определяющим таблицу локальных дескрипторов (LDT). Селектор LDT должен входить в таблицу глобальных дескрипторов. В качестве операнда команды lldt, содержащего селектор LDT, можно использовать 16- или 32-разрядный регистр общего назначения или 16- или 32-битовое поле памяти.

386P+ LMSW Загрузка слова состояния машины

Команда lmsw загружает в регистр слова состояния машины (так называется младшая половина управляющего регистра процессора CR0) слово состояния машины, взятое из указанного в команде операнда. В качестве операнда можно использовать 16- или 32-разрядный регистр общего назначения или 16- или 32-битовое поле памяти.

Команду lmsw можно использовать для перевода процессора из реального в защищенный режим или наоборот. В первом случае после чтения слова состояния командой smsw надо установить в нем бит 0 (бит PE) и загрузить назад в CR0 командой lmsw. Во втором случае после чтения слова состояния командой smsw надо сбросить в нем бит 0 и загрузить назад в CR0 командой lmsw.

LOCK Запирание шины

Префикс lock, помещенный перед командой; устанавливает сигнал на линии LOCK системной шины и запрещает доступ к шине другим процессорам на время выполнения данной команды. Этот префикс предназначен для использования в многопроцессорных многозадачных системах для обеспечения исключительного доступа к памяти данного процесса (и данного процессора) на время проверки или модификации некоторой ячейки памяти. Типичный пример операций такого рода – работа с семафорами.

LODS Загрузка строки
LODSB Загрузка строки по байтам
LODSW Загрузка строки по словам

Команды предназначены для операций над строками (строкой называется последовательность байтов или слов памяти с любым содержимым). Они загружают в регистр AL (в случае операций над байтами) или AX (в случае операций над словами) содержимое ячейки памяти по адресу, находящемуся в паре регистров DS:SI. После операции загрузки регистр SI получает положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера загружаемого элемента. Команда не имеет параметров и не воздействует на флаги процессора.

Вариант команды LODS имеет формат

```
lods   строка
```

(что не избавляет от необходимости инициализировать регистры DS:SI адресом строки; операнд лишь позволяет ассемблеру определить по описанию поля данных *строка* размерность загружаемых данных – байт или слово). В этом формате возможна замена сегмента строки *строка*:

```
lods   ES:строка
```

386+ LODSD Загрузка двойного слова из строки

Команда аналогична командам МП 86 lodb и lodsw, но позволяет загрузить из строки, адресуемой через регистры DS:ESI (DS:SI для 16-разрядных приложений), двойное слово в регистр EAX.

LOOP Циклическое выполнение, пока содержимое CX не равно нулю

Команда LOOP выполняет декремент содержимого регистра CX и, если оно не равно нулю, осуществляет переход на указанную метку вперед или назад в том же сегменте команд в диапазоне –128...+127 байт. Содержимое регистра CX рассматривается как целое число без знака, поэтому максимальное число повторений группы, включенных в цикл команд, составляет 65 536 (если перед входом в цикл CX=0). Команда не воздействует на флаги процессора.

386+ При использовании в качестве счетчика расширенного регистра ECX максимальное число шагов в цикле увеличивается до 2^{32}. Для того чтобы в 16-разрядном приложении процессор при выполнении команды loop использовал не 16-разрядный регистр CX, а 32-разрядный регистр ECX, перед командой loop необходимо указать префикс замены размера адреса 67h.

LOOPE Цикл, пока равно

Команда выполняет декремент содержимого регистра CX и, если оно не равно нулю и флаг ZF установлен, осуществляет переход на указанную метку вперед или назад в том же программном сегменте в диапазоне –128...+127 байт. Содержимое регистра CX рассматривается как целое число без знака, поэтому максимальное число повторений группы, включенных в цикл команд, составляет 65 536. Команда не воздействует на флаги процессора.

386+ При использовании в качестве счетчика расширенного регистра ECX максимальное число шагов в цикле увеличивается до 2^{32}. Для того чтобы в 16-разрядном приложении процессор при выполнении команд loope/loopz использовал не 16-разрядный регистр CX, а 32-разрядный регистр ECX, перед командами loope/loopz необходимо указать префикс замены размера адреса 67h.

LOOPNE Цикл, пока не равно

Команда выполняет декремент содержимого регистра CX и, если оно не равно нулю и флаг ZF сброшен, осуществляет переход на указанную метку вперед или назад в том же программном сегменте в диапазоне –128...+127 байт. Содержимое регистра CX рассматривается как целое число без знака, поэтому максимальное число повторений группы, включенных в цикл команд, составляет 65 536. Команда не воздействует на флаги процессора.

386+ При использовании в качестве счетчика расширенного регистра ECX максимальное число шагов в цикле увеличивается до 2^{32}. Для того чтобы в 16-разрядном приложении процессор при выполнении команд loopne/loopnz использовал не 16-

разрядный регистр CX, а 32-разрядный регистр ECX, перед командами loopne/loopnz необходимо указать префикс замены размера адреса 67h.

LOOPNZ Цикл, пока не нуль

Команда выполняет те же действия, что и LOOPNE.

LOOPZ Цикл, пока нуль

Команда выполняет те же действия, что и LOOPE.

386P+ LSL Загрузка границы сегмента

Команда lsl загружает в первый операнд границу сегмента из дескриптора сегмента, заданного селектором во втором операнде.

В качестве первого операнда команды lsl можно использовать 16- или 32-разрядный регистр общего назначения; в качестве второго – 16- или 32-разрядный регистр общего назначения или 16- или 32-битовое поле памяти.

386P+ LTR Загрузка регистра задачи TR

Команда ltr загружает регистр задачи TR селектором сегмента состояния задачи TSS из второго операнда, в качестве которого можно использовать 16- или 32-разрядный регистр общего назначения или 16- или 32-битовое поле памяти. Команда используется в защищенном режиме, если программный комплекс выполнен в виде нескольких самостоятельных задач, и переключения между ними осуществляются с использованием включенных в процессор аппаратных средств поддержки многозадачности.

MOV Пересылка данных

Команда MOV замещает первый операнд (приемник) вторым (источником). При этом исходное значение первого операнда теряется. В зависимости от описания операндов пересылается слово или байт. Если операнды описаны по-разному или режим адресации не позволяет однозначно определить размер операнда, для уточнения размера передаваемых данных в команду следует включить один из атрибутных операторов byte ptr или word ptr. Команда не воздействует на флаги процессора. В зависимости от использованных режимов адресации команда MOV осуществляет пересылки следующих видов:

- из регистра общего назначения в регистр общего назначения;
- из ячейки памяти в регистр общего назначения;
- из регистра общего назначения в ячейку памяти;
- непосредственный операнд в регистр общего назначения;
- непосредственный операнд в ячейку памяти;
- из регистра общего назначения в сегментные регистры;
- из сегментного регистра в регистр общего назначения;
- из сегментного регистра в ячейку памяти.

Запрещены пересылки из ячейки памяти в ячейку памяти (для этого предусмотрена команда MOVS), а также загрузка сегментного регистра непосредственным значением. Нельзя также непосредственно переслать содержимое одного сегментного регистра в другой.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

386P+ MOV Пересылка в\из специальных регистров

Этот вариант команды mov (с той же мнемоникой, но другими кодами операций) используется в защищенном режиме и предназначен для обмена данными со специальными регистрами процессора: управляющими CR0...CR3, тестирования TR6 и TR7, а также регистрами отладки DR0...DR7. Один из операндов команды mov должен быть 32-разрядным регистром общего назначения, другой – одним из специальных регистров процессора.

MOVS Пересылка данных из строки в строку
MOVSB Пересылка байта данных из строки в строку
MOVSW Пересылка слова данных из строки в строку

Команды предназначены для операций над строками (строкой называется последовательность байтов или слов памяти с любым содержимым). Они пересылают по одному элементу строки, который может быть байтом или словом. Первый операнд (приемник) адресуется через ES:DI, второй (источник) – через DS:SI. Операцию пересылки можно условно изобразить следующим образом:

```
(DS:SI) → (ES:DI)
```

После каждой операции пересылки регистры SI и DI получают положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера пересылаемых элементов.

Вариант команды MOVS имеет формат:

```
movs строка_1, строка_2
```

(что не избавляет от необходимости инициализировать регистры ES:DI и DS:SI адресами строк *строка_1* и *строка_2*; операнды лишь позволяют ассемблеру определить по описанию полей данных *строка_1* и *строка_2* размерность пересылаемых данных – байт или слово). В этом формате возможна замена сегмента второй строки:

```
movs   строка_1, ES:строка_2
```

Рассматриваемые команды могут предваряться префиксом повторения REP (повторять CX раз). После выполнения рассматриваемых команд регистры SI и DI указывают на ячейки памяти, находящиеся за теми (если DF=0) или перед теми (если DF=1) элементами строк, на которых закончились операции пересылки. Команды не воздействует на флаги процессора.

386+ MOVSD Пересылка двойного слова из строки в строку

Команда аналогична командам МП 86 movsb и movsw, но позволяет скопировать двойное слово из строки, адресуемой через регистры DS:ESI, в строку, адресуемую через регистры ES:EDI.

386+ MOVSX Пересылка с расширением знака

Команда пересылает байт в слово или двойное слово, а также слово в двойное слово с расширением знака. В качестве первого операнда (приемника) может использоваться 16- или 32-разрядный регистр общего назначения, в качестве второго – 8- или 16-разрядный регистр общего назначения или ячейка памяти такого же размера. Недопустима пересылка из памяти в память, в сегментный регистр или из него, а также непосредственного значения. Фактически команда movsx увеличивает размер как положительного, так и отрицательного числа, не изменяя ни его значения, ни знака.

386+ MOVZX Пересылка с расширением нуля

Команда пересылает байт в слово или двойное слово, а также слово в двойное слово с заполнением старших разрядов нулями. В качестве первого операнда (приемника) может использоваться 16- или 32-разрядный регистр общего назначения, в качестве второго – 8- или 16-разрядный регистр общего назначения или ячейка памяти такого же размера. Недопустима пересылка из памяти в память, в сегментный регистр или из него, а также непосредственного значения. Фактически команда movzx увеличивает размер числа, считая его числом без знака.

MUL Умножение целых беззнаковых чисел

Команда MUL выполняет умножение целого беззнакового числа, находящегося в регистре AL (в случае умножения на байт) или AX (в случае умножения на слово), на операнд-источник (целое число без знака). Размер произведения в два раза больше размера сомножителей.

Для 1-байтовых операций один из сомножителей помещается в регистр AL; после выполнения операции произведение записывается в регистр AX.

Для 2-байтовых операций один из сомножителей помещается в регистр AX; после выполнения операции произведение записывается в регистры DX:AX (в DX – старшая часть, в AX – младшая).

Если содержимое регистра AH после 1-байтового умножения или содержимое регистра DX после 2-байтового умножения не равны нулю, флаги CF и OF устанавливаются в 1. В противном случае оба флага сбрасываются в 0.

В качестве операнда-сомножителя можно указывать регистр данных или ячейку памяти; не допускается умножение на непосредственное значение.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров. При этом если указанный операнд представляет собой 32-байтовую величину, то результат размещается в регистрах EDX:EAX.

NEG Изменение знака, дополнение до 2

Команда NEG выполняет вычитание знакового целочисленного операнда из нуля, превращая положительное число в отрицательное и наоборот. В качестве операнда можно указывать регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Нельзя использовать в качестве операнда непосредственное значение. Команда воздействует на флаги OF, SF, ZF, AF, PF и CF.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

NOP Холостая команда

По команде NOP процессор не выполняет никаких действий, кроме увеличения на 1 (поскольку команда NOP занимает 1 байт) содержимого указателя команд IP. Команда иногда используется в отладочных целях чтобы "забить" какие-то ненужные команды, не изменяя длину загрузочного модуля или, наоборот, оставить место в загрузочном модуле для последующей вставки команд. В ряде случаев команды NOP включаются в текст объектного модуля транслятором. Команда не имеет ни параметров, ни операндов и не воздействует на флаги процессора.

NOT Инверсия, дополнение до 1

Команда NOT выполняет инверсию битов указанного операнда, заменяя 0 на 1 и наоборот. В качестве операнда можно указывать регистр (кроме сегментного) или

ячейку памяти размером байт или слово. Нельзя использовать в качестве операнда непосредственное значение. Команда не воздействует на флаги процессора.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

OR Логическое ВКЛЮЧАЮЩЕЕ ИЛИ

Команда OR выполняет операцию логического (побитового) сложения двух операндов. Результат замещает первый операнд (приемник); второй операнд (источник) не изменяется. В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами. Команда воздействует на флаги OF, SF, ZF, PF и CF, при этом флаги CF и OF всегда сбрасываются в 0.

Правила побитового сложения:

```
Первый операнд-бит  0 1 0 1
Второй операнд-бит  0 0 1 1
Бит результата      0 1 1 1
```

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

OUT Вывод в порт

Команда OUT выводит в порт, указываемый первым операндом, байт или слово соответственно из регистров AL или AX. Адрес порта помещается в регистр DX. Если адрес порта не превышает 255, он может быть указан непосредственным значением. Указание регистра-источника (AL или AH) обязательно. Команда не воздействует на флаги процессора.

386+
OUTS Вывод строки в порт
OUTSB Вывод байта в порт
OUTSW Вывод слова в порт
OUTSD Вывод двойного слова в порт

Команды предназначены для вывода данных в порт непосредственно из памяти. Адрес порта указывается, как и для команды out, в регистре DX, при этом задание адреса порта непосредственным значением не допускается. Данные извлекаются из памяти по адресу, находящемуся в паре регистров DS:ESI. Замена сегмента не допускается. Команда outsb передает в порт 1 байт, команда outsw – 1 слово, команда outsd – 1 двойное слово, а команда outs может быть использована для передачи байтов, слов и двойных слов. В последнем случае размер загружаемого данного определяется описанием строки (с помощью директив db, dw или dd). После передачи данных регистр ESI получает положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1, 2 или 4 в зависимости от размера передаваемых данных.

Вариант команды outs (не outsb, outsw или outsd) имеет формат

```
    outs   DX,строка
```

(что не избавляет от необходимости инициализировать регистры DS:ESI адресом строки).

Если устройство, адресуемое через порт, может принимать последовательность данных, то команды outs можно предварить префиксом повторения rep. В этом случае в порт пересылается CX элементов данных заданного размера.

Команды outs не воздействуют на флаги процессора.

POP Извлечение слова из стека

Команда POP пересылает слово из вершины стека (на которую указывает регистр SP) по адресу операнда-приемника. Затем содержимое SP увеличивается на 2 и указывает на новую вершину стека.

В качестве операнда-приемника можно использовать любой 16-разрядный регистр (кроме CS) или ячейку памяти. Команда не воздействует на флаги процессора.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

386+ POPA Восстановление из стека всех регистров

Команда popa восстанавливает из стека содержимое всех регистров, предварительно сохраненных в стеке командой pusha. Заполнение из стека регистров осуществляется в следующем порядке: DI, SI, BP, SP, BX, DX, CX, AX. Исходное содержимое указателя стека SP, сохраненное в стеке командой pusha, командой popa из стека извлекается, но отбрасывается. Команда не имеет параметров.

386+ POPAD Восстановление из стека всех регистров в 32-разрядном режиме

Команда popad восстанавливает из стека содержимое всех расширенных регистров, предварительно сохраненных в стеке командой pushad. Заполнение из стека регистров осуществляется в следующем порядке: EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX. Исходное содержимое указателя стека ESP, сохраненное в стеке командой pusha, командой popa из стека извлекается, но отбрасывается. Команда не имеет параметров.

POPF Восстановление из стека регистра флагов

Команда POPF пересылает определенные биты слова из вершины стека (на которую указывает регистр SP) в регистр флагов. Затем содержимое SP увеличивается на 2 и указывает на новую вершину стека. Команда воздействует на все флаги процессора.

+ POPFD Восстановление из стека расширенного регистра флагов

Команда popfd пересылает верхнее слово стека (на которое указывает регистр ESP) в расширенный регистр флагов EFLAGS. После этого содержимое ESP увеличивается на 4 и ESP указывает на предыдущее слово стека, которое теперь является его новой вершиной. Команда popfd не имеет параметров; она воздействует на все флаги процессора.

PUSH Занесение операнда в стек

Команда PUSH уменьшает на 2 содержимое указателя стека SP и заносит на эту новую вершину содержимое 2-байтового операнда-источника.

В качестве операнда-источника может использоваться любой 16-разрядный регистр (включая сегментные) или ячейка памяти. Не допускается занесение в стек непосредственного значения, хотя некоторые трансляторы преобразуют команду вида

```
push  1234h
```

в неэффективную последовательность операций со стеком, результатом которой будет проталкивание указанного операнда в стек. Команда не воздействует на флаги процессора.

386+ Допустима засылка в стек 32-битовых операндов (регистров и ячеек памяти), а также занесение в стек 8-, 16- и 32-битовых непосредственных значений. Каждое 8-

битовое значение занимает в стеке целое слово в 16-разрядных приложениях и два слова в 32-разрядных. Операнды любого допустимого размера могут заноситься в стек вперемежку, если это не вступает в противоречие с операциями по извлечению этих данных из стека.

386+ PUSHA Сохранение в стеке всех регистров

Команда pusha сохраняет в стеке содержимое всех регистров в следующем порядке: AX, CX, DX, BX, значение указателя стека SP перед выполнением данной команды, далее BP, SI и DI. Команда не имеет параметров и не воздействует на флаги процессора.

386+ PUSHAD Сохранение в стеке всех регистров в 32-разрядном режиме

Команда pushad сохраняет в стеке содержимое всех регистров в следующем порядке: EAX, ECX, EDX, EBX, значение указателя стека ESP перед выполнением данной команды, далее EBP, ESI и EDI. Команда не имеет параметров и не воздействует на флаги процессора.

PUSHF Занесение в стек содержимого регистра флагов

Команда PUSH уменьшает на 2 содержимое указателя стека SP и заносит на эту новую вершину содержимое регистра флагов.

386+ PUSHFD Занесение в стек содержимого расширенного регистра флагов

Команда pushfd уменьшает на 4 содержимое указателя стека ESP и заносит на эту новую вершину содержимое расширенного регистра флагов EFALGS. При этом сохраняются все флаги процессора. Команда pushfd не имеет параметров и не воздействует на флаги процессора.

RCL Циклический сдвиг влево через бит переноса

Команда RCL осуществляет сдвиг влево всех битов операнда. Если команда записана в формате

```
RCL     операнд,1
```

выполняется сдвиг на 1 бит. В младший бит операнда заносится значение флага CF; старший бит операнда загружается в CF. Если команда записана в формате

```
RCL     операнд,CL
```

сдвиг осуществляется на число бит, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов старшие биты операнда поступают сначала в CF, а оттуда – в младшие биты операнда (рис. П-1.3).

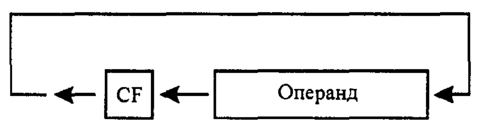


Рис. П-1.3. Действие команды rcl

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. В качестве операнда нельзя использовать непосредственное значение. Команда воздействует на флаги OF и CF.

386+ Допустим сдвиг 32-битовых операндов. Допустимо указание числа битов сдвига как с помощью регистра CL, так и непосредственным значением. Максимальная величина сдвига составляет 31 бит.

RCR Циклический сдвиг вправо через бит переноса

Команда RCR осуществляет сдвиг вправо всех битов операнда. Если команда записана в формате

```
RCR    операнд,1
```

сдвиг осуществляется на 1 битов. В старший бит операнда заносится значение флага CF; младший бит операнда загружается в CF. Если команда записана в формате

```
RCL    операнд,CL
```

сдвиг осуществлястся на число бит, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов младшие биты операнда поступают сначала в CF, а оттуда – в старшие биты операнда (рис. П-1.4).

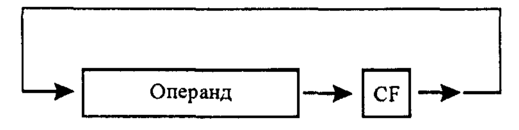


Рис. П-1.4. Действие команды rcr

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. В качестве операнда нельзя использовать непосредственное значение. Команда воздействует на флаги OF и CF.

386+ Допустим сдвиг 32-битовых операндов. Допустимо указание числа битов сдвига как с помощью регистра CL, так и непосредственным значением. Максимальная величина сдвига составляет 31 бит.

Pentium+P RDMSR Чтение особого регистра модели

Команда читает содержимое внутреннего регистра, специфического для конкретной модели процессора.

REP Повторение
REPE Повторение пока равно
REPZ Повторение пока нуль
REPNE Повторение пока равно
REPNZ Повторение пока не равно

Префиксы повторения позволяют организовать циклическое выполнение команд обработки строк cmps, movs и scas и при этом проверять наличие указанного в префиксе условия.

Префикс rep, будучи установлен перед строковыми командами movs или stos, заставляет их выполняться CX раз.

Префикс repe (и полностью эквивалентный ему префикс repz), будучи установлен перед строковыми командами cmps или scas, заставляет их выполняться до тех пор, пока результат выполнения равен нулю и, соответственно, ZF=1, но не более CX раз.

Префикс repne (и полностью эквивалентный ему префикс repnz), будучи установлен перед строковой командой cmps или scas, заставляет ее выполняться до тех пор, пока результат выполнения не равен нулю и, соответственно, ZF=0, но не более CX раз.

RET Возврат из процедуры
RETN Возврат из ближней процедуры
RETF Возврат из дальней процедуры

Команда RET извлекает из стека адрес возврата и передает управление в программу, вызвавшую процедуру. Если командой RET завершается ближняя процедура, объ-

явленная с атрибутом NEAR, или используется форма команды RETN, со стека снимается одно слово – смещение точки возврата. Передача управления в этом случае осуществляется в пределах одного сегмента. Если командой RET завершается дальняя процедура, объявленная с атрибутом FAR, или используется форма команды RETF, со стека снимаются два слова – смещение и сегмент точки возврата. В этом случае передача управления будет межсегментной.

В команду RET может быть включен необязательный операнд (кратный двум), который указывает, на сколько байтов дополнительно смещается (в сторону больших адресов) указатель стека после возврата в вызывающую программу. Прибавляя эту константу к новому значению SP, команда RET обходит аргументы, помещенные в стек вызывающей программой перед вызовом команды CALL. Обе разновидности команды не воздействуют на флаги процессора.

ROL Циклический сдвиг влево

Команда ROL осуществляет сдвиг влево всех битов операнда. Если команда записана в формате

```
ROL    операнд,1
```

сдвиг осуществляется на 1 бит. Старший бит операнда загружается в его младший разряд. Если команда записана в формате

```
ROL    операнд,CL
```

сдвиг осуществляется на число бит, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов старшие биты операнда перемещаются в его младшие разряды (рис. П-1.5).

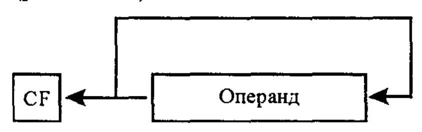


Рис. П-1.5. Действие команды rol

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Не допускается использовать в качестве операнда непосредственное значение. Команда воздействует на флаги OF и CF.

386+ Допустим сдвиг 32-битовых операндов. Допустимо указание числа бит сдвига как с помощью регистра CL, так и непосредственным значением. Максимальная величина сдвига составляет 31 бит.

ROR Циклический сдвиг вправо

Команда ROR осуществляет циклический сдвиг вправо всех битов операнда. Если команда записана в формате

```
ROR    операнд,1
```

сдвиг осуществляется на 1 бит. Младший бит операнда записывается в его старший разряд. Если команда записана в формате

```
ROR    операнд,CL
```

сдвиг осуществляется на число битов, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов младшие биты операнда перемещаются в его старшие разряды (рис. П-1.6).

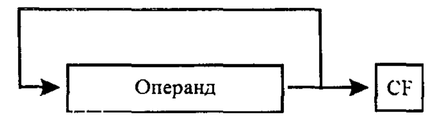


Рис. П-1.6. Действие команды ror

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Нельзя использовать в качестве операнда непосредственное значение. Команда воздействует на флаги OF и CF.

386+ Допустим сдвиг 32-битовых операндов. Допустимо указание числа битов сдвига как с помощью регистра CL, так и непосредственным значением. Максимальная величина сдвига составляет 31 бит.

SAHF Запись содержимого регистра AH в регистр флагов

Команда SAHF копирует биты 7, 6, 4, 2 и 0 регистра AH в младший байт регистра флагов, влияя на флаги SF, ZF, AF, PF и CF.

Команда SAHF (совместно с командой LAHF) дает возможность читать и изменять значение флагов процессора, в том числе флагов SF, ZF, AF и PF, которые нельзя изменить непосредственно. Однако следует иметь в виду, что команда sahf заполняет только младший байт регистра флагов. Поэтому нельзя изменить с ее помощью, например, состояние флага OF.

SAL Арифметический сдвиг влево

Команда SAL осуществляет сдвиг влево всех битов операнда. Старший бит операнда поступает в флаг CF. Если команда имеет форму

```
SAL     операнд,1
```

сдвиг осуществляется на 1 бит. В младший бит операнда загружается 0. Если команда записана в формате

```
SAL     операнд,CL
```

сдвиг осуществляется на число битов, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов старшие биты операнда, пройдя через флаг CF, теряются, а младшие заполняются нулями (рис. П-1.7).

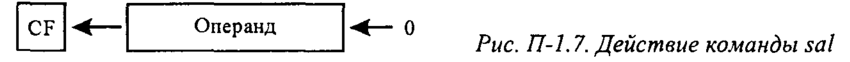


Рис. П-1.7. Действие команды sal

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Нельзя использовать в качестве операнда непосредственное значение.

Каждый сдвиг влево эквивалентен умножению знакового числа на 2, поэтому команду SAL удобно использовать для возведения операнда в степень 2. Команда воздействует на флаги OF, SF, ZF, PF и CF.

386+ Допустим сдвиг 32-битовых операндов. Допустимо указание числа бит сдвига как с помощью регистра CL, так и непосредственным значением. Максимальная величина сдвига составляет 31 бит.

SAR Арифметический сдвиг вправо

Команда SAR осуществляет сдвиг вправо всех битов операнда. Младший битов операнда поступает в флаг CF. Если команда записана в формате

```
SAR     операнд,1
```

сдвиг осуществляется на 1 бит. Старший бит операнда сохраняет свое значение. Если команда записана в формате

```
SAR    операнд,CL
```

сдвиг осуществляется на число битов, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов младшие биты операнда, пройдя через флаг CF, теряются, а старший бит расширяется вправо (рис. П-1.8).

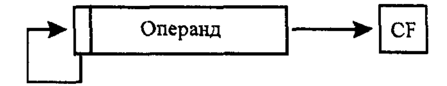


Рис. П-1.8. Действие команды sar

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Нельзя использовать в качестве операнда непосредственное значение. Команда воздействует на флаги OF, SF, ZF, AF, PF и CF.

Каждый сдвиг вправо эквивалентен делению знакового числа на 2, поэтому команду SAR удобно использовать для деления операнда на целые степени двух.

386+ Допустим сдвиг 32-битовых операндов. Допустимо указание числа битов сдвига как с помощью регистра CL, так и непосредственным значением. Максимальная величина сдвига составляет 31 бит.

SBB Целочисленное вычитание с займом

Команда SBB вычитает второй операнд (источник) из первого (приемника). Результат замещает первый операнд, предыдущее значение которого теряется. Если установлен флаг CF, из результата вычитается еще 1. Таким образом, если команду вычитания записать в общем виде

```
sbb    операнд_1, операнд_2
```

то ее действие можно условно изобразить следующим образом:

```
операнд_1 - операнд_2 - CF → операнд_1
```

В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Команда воздействует на флаги OF, SF, ZF, PF и CF.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

SCAS Сканирование строки с целью сравнения

SCASB Сканирование строки байтов с целью сравнения

SCASW Сканирование строки слов с целью сравнения

Команды предназначены для операций над строками (строкой называется последовательность байтов или слов памяти с любым содержимым). Они сравнивают содержимое регистров AL (в случае операций над байтами) или AX (в случае операций над словами) с содержимым ячейки памяти по адресу, находящемуся в паре регистров ES:DI. Операция сравнения осуществляется путем вычитания содержимого ячейки памяти из содержимого AL или AX. Результат операции воздействует на регистр флагов, но не изменяет ни один из операндов. Таким образом, операцию сравнения можно условно изобразить следующим образом:

```
AX или AL - (ES:DI) → флаги процессора
```

После каждой операции сравнения регистр DI получает положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера сравниваемых элементов.

Вариант команды SCAS имеет формат

```
scas    строка
```

(что не избавляет от необходимости инициализировать регистры ES:DI адресом строки *строка*; операнд лишь позволяет ассемблеру определить по описанию поля данных *строка* размерность сравниваемых данных – байт или слово). Замена сегментного регистра (ES), через который адресуется строка, невозможна. Команды воздействуют на флаги OF, SF, ZF, AF, PF и CF.

Рассматриваемые команды могут предваряться префиксами повторения REPE/REPZ (повторять до первого неравенства) и REPNE/REPNZ (повторять до первого равенства). В любом случае выполняется не более CX операций над последовательными элементами.

После выполнения рассматриваемых команд регистр DI указывает на ячейку памяти, находящуюся за тем (если DF=0) или перед тем (если DF=1) элементом строки, на котором закончились операции сравнения.

386+ SCASD Сканирование строки двойных слов с целью сравнения

Команда аналогична командам МП 86 scab и scasw, но позволяет сравнивать содержимое расширенного регистра EAX с двойным словом в памяти, адресуемым через регистры ES:EDI.

386+ SETcc Установка байта по условию

Команды, обозначаемые (в книгах, не в программах!) SETcc, осуществляют запись в указанный байтовый операнд 1 или 0 в зависимости от одного из 16 условий, определяемых флагами состояния. Если условие cc выполняется, команда записывает в операнд 1; если условие не выполняется – 0. В качестве операнда можно использовать байтовый регистр или 8-битовую ячейку памяти.

Команды условной установки байта, предусмотренные в составе команд процессора, приведены в табл. П-1.2.

Таблица П-1.2. Команды условной установки байта

Команда	*Установить 1, если*	*Условие установки 1*
seta	выше	CF=0 и ZF=0
setae	выше или равно	CF=0
setb	ниже	CF=1
setbe	ниже или равно	CF=1 или ZF=1
setc	перенос	CF=1
sete	равно	ZF=1
setg	больше	ZF=0 или SF=OF
setge	больше или равно	SF=OF
setl	меньше	SF не равно OF
setle	меньше или равно	ZF=1 или SF не равно OF
setna	не выше	CF=1 или ZF=1
setnae	не выше и не равно	CF=1
setnb	не ниже	CF=0
setnbe	не ниже и не равно	CF=0 и ZF=0
setnc	нет переноса	CF=0

setne	не равно	ZF=0
setng	не больше	ZF=1 или SF не равно OF
setnge	не больше и не равно	SF не равно OF
setnl	не меньше	SF=OF
setnle	не меньше и не равно	ZF=0 и SF=OF
setno	нет переполнения	OF=0
setnp	нет четности	PF=0
setns	знаковый бит равен 0	SF=0
setnz	не нуль	ZF=0
seto	переполнение	OF=1
setp	есть четность	PF=1
setpe	сумма битов четная	PF=1
setpo	сумма битов нечетная	PF=0
sets	знаковый бит равен	SF=1
setz	нуль	ZF=1

Команды, осуществляющие установку по условию "выше – ниже", предназначены для анализа чисел без знака; команды, осуществляющие установку по условию "больше – меньше", предназначены для анализа чисел со знаком.

386P+ SGDT Сохранение в памяти содержимого регистра таблицы глобальных дескрипторов

Команда копирует содержимое регистра таблицы глобальных дескрипторов GDTR (линейный базовый адрес таблицы и ее границу) в поле из 6 байт, указанное в качестве операнда.

SHL Логический сдвиг влево

Команда SHL выполняет те же действия, что и SAL.

386+ SHLD Логический сдвиг влево двойного слова

Трехоперандная команда shld с операндами op1, op2 и op3 осуществляет сдвиг влево первого из своих операндов op1. Число битов сдвига определяется третьим операндом op3. По мере сдвига операнда op1 влево выдвигаемые из него старшие биты, пройдя через флаг CF, теряются, а на освобождающиеся места со стороны его младших битов поступают старшие биты второго операнда op2, как если бы он вдвигался своим левым (старшим) концом в op1. Однако после завершения сдвига значение операнда op2 не изменяется (рис. П-1.9). Во флаге CF остается последний выдвинутый из операнда op1 бит. Максимальное число битов сдвига составляет 31.

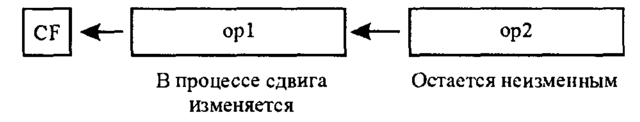


Рис. П-1.9. Действие команды shld

В качестве первого операнда op1 можно указывать 16- или 32-разрядный регистр общего назначения или 16- или 32-битовую ячейку памяти. Вторым операндом op2 может служить только 16- или 32-разрядный регистр общего назначения. Третий операнд, характеризующий число битов сдвига, может находиться в регистре CL или быть непосредственным значением.

Команда воздействует на флаги OF, SF, ZF, PF и CF.

SHR Логический сдвиг вправо

Команда SHR осуществляет сдвиг вправо всех битов операнда. Младший бит операнда поступает в флаг CF. Если команда имеет форму

```
SHR     операнд,1
```

сдвиг осуществляется на 1 бит. В старший бит операнда загружается 0, а младший теряется. Если команда имеет форму

```
SHR     операнд,CL
```

сдвиг осуществляется на число битов, указанное в регистре-счетчике CL, при этом в процессе последовательных сдвигов старшие биты операнда заполняются нулями, а младшие, пройдя через флаг CF, теряются (рис. П-1.10).

0 → Операнд → CF

Рис. П-1.10. Действие команды shr

В качестве операнда можно указывать любой регистр (кроме сегментного) или ячейку памяти размером как в байт, так и в слово. Нельзя использовать в качестве операнда непосредственное значение. Команда воздействует на флаги CF, OF, PF, SF и ZF.

386+ Допустим сдвиг 32-битовых операндов. Допустимо указание числа битов сдвига как с помощью регистра CL, так и непосредственным значением. Максимальная величина сдвига составляет 31 бит.

386+ SHRD Логический сдвиг вправо двойного слова

Трехоперандная команда shrd с операндами op1, op2 и op3 осуществляет сдвиг вправо первого из своих операндов op1. Число битов сдвига определяется третьим операндом op3. По мере сдвига операнда op1 вправо выдвигаемые из него младшие биты, пройдя через флаг CF, теряются, а на освобождающиеся места со стороны его старших битов поступают младшие биты второго операнда op2, как если бы он вдвигался своим правым (младшим) концом в op1. Однако после завершения сдвига значение операнда op2 не изменяется (рис. П-1.11). Во флаге CF остается последний выдвинутый из операнда op1 бит. Максимальное число битов сдвига составляет 31.



Рис. П-1.11. Действие команды shrd

В качестве первого операнда op1 можно указывать 16- или 32-разрядный регистр общего назначения или 16- или 32-битовую ячейку памяти. Вторым операндом op2 может служить только 16- или 32-разрядный регистр общего назначения. Третий операнд, характеризующий число битов сдвига, может находиться в регистре CL или быть непосредственным значением.

Команда воздействует на флаги OF, SF, ZF, PF и CF.

386P+ SIDT Сохранение в памяти содержимого регистра таблицы дескрипторов прерываний

Команда копирует содержимое регистра таблицы дескрипторов прерываний IDTR (линейный базовый адрес таблицы и ее границу) в поле из 6 байт, указанное в качестве операнда.

386P+ SLDT Сохранение содержимого регистра таблицы локальных дескрипторов

Команда копирует содержимое регистра таблицы локальных дескрипторов LDTR (селектор таблицы) в 16- или 32-разрядный регистр или в 16- или 32-битовое поле памяти, указанные в качестве операнда.

386P+ SMSW Сохранение слова состояния машины

Команда smsw считывает слово состояния машины (так называется младшая половина управляющего регистра процессора CR0) и загружает его в указанный в команде 16-разрядный регистр общего назначения или в 16-битовое поле памяти.

Команду smsw можно использовать для перевода процессора из реального в защищенный режим или наоборот. В первом случае после чтения слова состояния командой smsw надо установить в нем бит 0 (бит PE) и загрузить назад в CR0 командой lmsw. Во втором случае после чтения слова состояния командой smsw надо сбросить в нем бит 0 и загрузить назад в CR0 командой lmsw.

STC Установка флага переноса

Команда STC устанавливает флаг переноса CF в регистре флагов. Команда не имеет параметров и не воздействует на остальные флаги процессора.

STD Установка флага направления

Команда STD устанавливает флаг направления DF в регистре флагов, определяя тем самым обратное направление выполнения строковых операций (в порядке убывания адресов элементов строки). Команда не имеет параметров и не воздействует на остальные флаги процессора.

STI Установка флага прерывания

Команда STI устанавливает флаг IF в регистре флагов, разрешая все аппаратные прерывания. Команда не имеет параметров и не воздействует на остальные флаги процесссора.

STOS Запись в строку данных
STOSB Запись байта в строку данных
STOSW Запись слова в строку данных

Команды предназначены для операций над строками (строкой называется последовательность байтов или слов с любым содержимым). Они копируют содержимое регистра AL (при операциях над байтами) или AX (при операциях над словами) в ячейку памяти соответствующего размера по адресу, определяемому содержимым ES:DI. После операции копирования DI получает положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера копируемого элемента.

Вариант команды STOS имеет формат

```
stos   строка
```

(что не избавляет от необходимости инициализировать регистры ES:DI адресом строки *строка*; операнд лишь позволяет ассемблеру определить по описанию поля данных *строка* размерность записываемых данных – байт или слово). Заменить сегментный регистр ES нельзя. Команды не воздействует на флаги процессора.

Рассматриваемые команды могут предваряться префиксом повторения REP. В этом случае они повторяются CX раз, заполняя последовательные ячейки памяти одним и тем же содержимым регистров AL или AX.

386+ STOSD Запись двойного слова в строку данных

Команда аналогична командам МП 86 stosb и stosw, но позволяет записать в строку, адресуемую через регистры ES:EDI, двойное слово из регистра EAX.

386P+ STR Сохранение содержимого регистра состояния задачи

Команда str копирует содержимое регистра задачи TR (селектор сегмента состояния задачи) в 2-байтовый регистр общего назначения или 16-битовую ячейку памяти, указанные в качестве операнда.

SUB Вычитание целых чисел

Команда SUB вычитает второй операнд (источник) из первого (приемника) и помещает результат на место первого операнда. Исходное значение первого операнда (уменьшаемое) теряется. Таким образом, если команду вычитания записать в общем виде

```
sub    операнд_1, операнд_2
```

то ее действие можно условно изобразить следующим образом:

```
операнд_1 - операнд_2 → операнд_1
```

В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго операнда – еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака. Команда воздействует на флаги OF, SF, ZF, AF, PF и CF.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

TEST Логическое сравнение

Команда TEST выполняет операцию логического И над двумя операндами и в зависимости от результата устанавливает флаги SF, ZF и PF. Флаги OF и CF сбрасываются, а AF имеет неопределенное значение. Состояние флагов можно затем проанализировать командами условных переходов. Команда TEST не изменяет ни одного из операндов.

В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда – еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами и представлять числа со знаком или без знака.

Правила побитового умножения:

```
Первый операнд-бит  0 1 0 1
Второй операнд-бит  0 0 1 1
Бит результата      0 0 0 1
```

Флаг SF устанавливается в 1, если в результате выполнения команды образовалось число с установленным знаковым битом.

Флаг ZF устанавливается в 1, если в результате выполнения команды образовалось число, состоящее из одних двоичных нулей.

Флаг PF устанавливается в 1, если в результате выполнения команды образовалось число с четным количеством двоичных единиц в его битах.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

386P+ VERR Проверка сегмента на чтение

Команда verr позволяет определить, разрешено ли чтение из сегмента, за которым закреплен селектор, передаваемый команде в качестве ее операнда. Операндом может служить 16-разрядный регистр общего назначения или 16-битовая ячейка памяти.

386P+ VERW Проверка сегмента на запись

Команда verw позволяет определить, разрешена ли запись в сегмент, за которым закреплен селектор, передаваемый команде в качестве ее операнда. Операндом может служить 16-разрядный регистр общего назначения или 16-битовая ячейка памяти.

486+ XADD Обмен и сложение

Команда xadd выполняет в одной операции сложение и обмен операндов. Команда требует двух операндов, причем первый операнд должен быть ячейкой памяти, а второй – регистром. После сложения операндов исходное содержимое памяти переносится во второй операнд (регистр), а полученная сумма записывается в память (на место первого слагаемого) (рис. П-1.12). Команда воздействует на флаги OF, SF, ZF, AF, PF и CF.

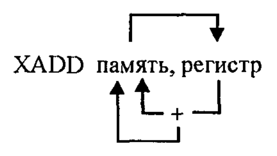


Рис. П-1.12. Действие команды xadd

XCHG Обмен данными между операндами

Команда XCHG пересылает значение первого операнда во второй, а второго – в первый. В качестве операндов можно указывать регистр (кроме сегментного) или ячейку памяти, а в качестве второго операнда – еще и непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами со знаком или без знака. Команда не воздействует на флаги процессора.

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

XLAT Табличная трансляция

Команда XLAT осуществляет выборку байта из таблицы. В регистре BX должен находиться относительный адрес таблицы, а в регистре AL – смещение в таблице к выбираемому байту (его индекс). Выбранный байт загружается в регистр AL, замещая находившееся в нем смещение. Длина таблицы может достигать 256 байт. Команда XLAT не имеет явных операндов, но требует предварительной настройки регистров BX и AL. Команда не воздействует на флаги процессора.

386+ XLAT Табличная трансляция

386+ XLATB Табличная трансляция

Команда xlatb эквивалентна команде xlat МП 86 за исключением того, что для 32-разрядных приложений относительный адрес таблицы размещается в расширенном регистре EBX.

Команда xlat может иметь в качестве операнда относительный адрес таблицы трансляции; в этом случае помещение адреса таблицы в регистр EBX не требуется. Действие команды от этого не изменяется, однако возможна замена сегмента.

XOR Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ

Команда XOR выполняет операцию логического (побитового) ИСКЛЮЧАЮЩЕГО ИЛИ над двумя операндами. Результат операции замещает первый операнд. Каждый бит результата устанавливается в 1, если соответствующие биты операндов различны, и сбрасывается в 0, если соответствующие биты операндов совпадают.

В качестве первого операнда можно указывать регистр (кроме сегментного) или ячейку памяти, в качестве второго – регистр (кроме сегментного), ячейку памяти или непосредственное значение, однако не допускается определять оба операнда одновременно как ячейки памяти. Операнды могут быть байтами или словами. Команда воздействует на флаги OF, SF, ZF, PF и CF, причем флаги OF и CF всегда сбрасываются, а остальные флаги устанавливаются в зависимости от результата.

Правила побитового ИСКЛЮЧАЮЩЕГО ИЛИ:

```
Первый операнд-бит  0 1 0 1
Второй операнд-бит  0 0 1 1
Бит результата      0 1 1 0
```

386+ Допустимо использование 32-битовых операндов и дополнительных режимов адресации 32-разрядных процессоров.

Основные директивы ассемблера TASM

Директивы определения данных

DB Определение данных размером в байт.

DW Определение данных размером в слово.

DD Определение данных размером в двойное слово.

DQ Определение данных размером в четверное слово.

DF Определение данных размером в 6 байт.

DP Определение данных размером в 6 байт.

DT Определение данных размером в 10 байт.

Все вышеперечисленные директивы резервируют и заполняют данными требуемое число полей, размер которых соответствует конкретной директиве. Формат директив (на примере директивы DB):

```
имя DB данное или перечень данных через запятую
```

Директивы присваивания

EQU Присвоение значения указанному идентификатору. Формат директивы:

```
имя EQU выражение
```

Идентификатор *имя* получает значение указанного в директиве выражения *выражение*. Ассемблер по ходу трансляции программы заменяет все вхождения в программу идентификатора *имя* значением выражения *выражение*. Идентификатор, определенный с помощью директивы EQU, переопределить нельзя.

= Присвоение значения указанному идентификатору. Формат директивы:

```
имя = выражение
```

Идентификатор *имя* получает значение указанного в директиве выражения *выражение*. Ассемблер по ходу трансляции программы заменяет все вхождения в программу идентификатора *имя* значением выражения *выражение*. В отличие от директивы EQU, директива = позволяет в любой точке программы присвоить идентификатору *имя* другое значение.

Директивы счетчика текущего адреса

$ Идентификатор счетчика текущего адреса. Символическое обозначение ячейки в трансляторе, в которой по мере трансляции каждого сегмента наращивается текущее значение смещения в этом сегменте.

LABEL Задание значения идентификатора (имени поля данных или программной метки). Присваивает текущее значение счетчика текущего адреса указанному идентификатору. Может использоваться в сегменте команд для определения программной метки требуемой дальности или в сегменте данных для задания альтернативного имени поля данных.

ORG Установка значения счетчика текущего адреса. Формат директивы:

```
ORG выражение
```

Устанавливает указанное значение в счетчике текущего адреса для текущего сегмента (команд или данных). При установке значения, большего, чем текущее, в памяти резервируется указанное число байтов; при установке значения меньшего, чем текущее, счетчик текущего адреса возвращается к указанному адресу. Допустимо последовательное многократное использование этой директивы.

Операторы типа

PTR Задание типа переменной или метки. Формат директивы:

```
тип PTR выражение
```

Оператор PTR заставляет ассемблер на время текущей операции считать, что выражение имеет тип тип (хотя, возможно, объявлено оно было по-другому). В качестве выражения может быть использован любой операнд. Тип может принимать форму BYTE, WORD, DWORD, FWORD, QWORD или TBYTE для операндов, находящихся в памяти, а также форму NEAR, FAR или PROC для меток или имен процедур.

OFFSET Возвращает смещение (младшую часть адреса). Формат директивы:

```
OFFSET выражение
```

SEG Возвращает сегмент (старшую часть адреса). Формат директивы:

```
SEG выражение
```

Директивы организации сегментов и процедур

SEGMENT Открытие сегмента памяти.

ENDS Закрытие сегмента памяти. Формат директив:

```
имя SEGMENT
тело сегмента
имя ENDS
```

Директива segment открывает новый или продолжает ранее объявленный сегмент (команд, данных или стека). Результат трансляции всех последующих предложений до директивы ends будет включен в этот сегмент.

PROC Указание начала процедуры.

ENDP Указание конца процедуры. Формат директив:

```
имя PROC [NEAR | FAR]
Предложения, составляющие процедуру
имя ENDP
```

Программные строки, заключенные между операторами proc и endp, считаются принадлежащими процедуре с указанным именем. Спецификатор дальности указывает, является ли процедура ближней (NEAR) или дальней (FAR). Все команды ret (но не retf) внутри процедуры автоматически принимают дальность, соответствующую указанному спецификатору. При отсутствии спецификатора дальности ассемблер считает процедуру ближней. Имя процедуры при операторе ENDP можно опустить.

ASSUME Задание сегментного регистра, используемого по умолчанию. Формат директивы:

```
ASSUME сегм_регстр:сегмент [[,сегм_регстр:сегмент]...]
```

Директива ASSUME связывает указанные в ней сегментные регистры с соответствующими сегментами. Это даст возможность в программе не указывать явным образом при каждом обращении к ячейке памяти сегментный регистр, через который надлежит обращаться к этой ячейке. Директива ASSUME не инициализирует сегментные регистры. Программа обязана перед первым обращением к любой ячейке памяти инициализировать какой-либо сегментный регистр, т. е. заслать в него базовый адрес соответствующего сегмента. Дальнейшее обращение к ячейкам памяти этого сегмента должно, вообще говоря, выполняться с явным указанием инициализированного сегментного регистра:

```
        inc    ES:mem
```

Связывание сегментного регистра с сегментом с помощью директивы ASSUME позволяет опустить обозначение сегментного регистра в командах обращения к памяти:

```
        inc    mem
```

Допускается изменять связывание любого сегментного регистра по ходу выполнения программы, если один и тот же сегментный регистр будет использоваться для последовательного обращения к различным сегментам памяти.

Связывание сегментного регистра CS с сегментом команд является обязательным. Если программа состоит из нескольких сегментов команд, перед каждым сегментом команд должна стоять директива

```
        assume CS:code_seg
```

где *code_seg* – имя очередного сегмента команд.

Если в команде обращения к памяти не указан сегментный регистр, то процессор выполняет эту команду, используя регистр DS. Поэтому, как правило, связывание требуется только для остальных сегментных регистров данных: ES, FS и GS.

Директивы описания макросов

MACRO Начало определения макроса.

ENDM Конец определения макроса. Формат директив:

```
имя MACRO [параметр[,параметр]...]
Предложения макроса
ENDM
```

LOCAL Объявление временных имен меток, заменяемых в процессе повторных макрорасширений на уникальные имена. Формат директивы:

```
LOCAL метка[,метка]...
```

Ассемблер в процессе макрорасширения макроса подставляет на место локальной метки *метка* идентификатор вида *??число*. Если в программе данный макрос вызывается неоднократно, при каждом следующем макрорасширении числовая часть метки будет получать приращение 1.

Директива повторения

REPT Повторение предложений языка. Формат директивы:

```
REPT выражение
Предложения языка
ENDM
```

Выражение, рассматриваемое как целое число, задает число повторений блока *Предложения языка*. Внутри блока повторения могут находиться любые предложения языка ассемблера.

Директивы организации листинга

.XLIST Подавление дальнейшего вывода в листинг трансляции.

.LIST Разрешение дальнейшего вывода в листинг трансляции.

.SALL Подавление вывода в листинг трансляции текстов макрорасширений.

Директивы условной трансляции

IF Трансляция по заданному условию. Формат директивы:

```
IF условие
Программные предложения 1
[ELSE
Программные предложения 2]
ENDIF
Программные предложения 3
```

Если условие выполняется, то транслируются *Программные предложения 1*. В противном случае транслируются *Программные предложения 2*. В любом случае далее транслируются *Программные предложения 3*.

IFDEF Трансляция при условии, что идентификатор определен. Формат директивы:

```
IFDEF имя
Программные предложения
ENDIF
```

Трансляция строк *Программные предложения* осуществляется только если идентификатор *имя* уже был определен в программе.

IFNDEF Трансляция при условии, что идентификатор не определен. Формат директивы:

```
IFNDEF имя
Программные предложения
ENDIF
```

Трансляция строк *Программные предложения* осуществляется, только, если идентификатор *имя* еще не был определен в программе.

Директивы организации структуры программы

.MODEL Задание модели памяти. Директива .MODEL имеет большое количество параметров и широкие возможности управления структурой программы и составляющих ее процедур, многие из которых используются относительно редко. Ниже приводится неполное описание этой директивы с указанием наиболее существенных деталей. Формат директивы:

```
.MODEL [разрядность] модель [язык]
```

Параметр *разрядность* может принимать следующие значения:

USE16 – все сегменты программы будут 16-разрядными;

USE32 – все сегменты программы будут 32-разрядными.

Параметр *модель* может принимать следующие значения:

SMALL (малая модель памяти) – приложение может включать лишь по одному сегменту команд, данных и стека; при этом, поскольку данные и стек входят в общую группу, их суммарный размер не должен превышать 64 Кбайт. Сегмент команд также не должен быть больше 64 Кбайт. Наиболее широко распространенная модель для простых программ;

MEDIUM (средняя модель памяти) – приложение может иметь несколько сегментов команд и по одному сегменту данных и стека. Сегменты команд выделяются с помощью директивы .CODE с указанием для каждого сегмента уникального имени. Как и в случае малой модели, данные и стек входят в общую группу и их суммарный размер не должен превышать 64 Кбайт. Модель используется в тех случаях, когда приложение при относительно небольшом объеме данных (менее 64 Кбайт) содержит сложные вычислительные алгоритмы, которые нельзя уместить в 64 Кбайт. Чаще всего в этом случае в дополнительные сегменты команд выносят процедуры-подпрограммы, обращение к которым осуществляется с помощью команд дальних вызовов;

COMPACT (компактная модель памяти) – приложение может иметь лишь один сегмент команд размером не более 64 Кбайт, сегмент стека и несколько сегментов данных, каждый размером не более 64 Кбайт. Сегменты данных выделяются с помощью директивы .FARDATA с указанием для каждого сегмента уникального имени. Модель используется в тех случаях, когда приложение должно обрабатывать большие объемы данных. В этом случае данные следует разбить на массивы размером не более 64 Кбайт каждый и разнести их по разным сегментам данных;

LARGE (большая модель памяти) – приложение может иметь несколько сегментов команд и несколько сегментов данных. Сегменты команд выделяются с помощью директивы .CODE с указанием для каждого сегмента уникального имени. Сегменты данных выделяются с помощью директивы .FARDATA с указанием для каждого сегмента

уникального имени. Модель используется в тех случаях, когда приложение содержит сложные вычислительные алгоритмы, которые нельзя уместить в 64 Кбайт, и должно обрабатывать большие объемы данных. Замечания относительно использования пакетов MASM и TASM см. в описании параметра COMPACT;

HUGE (огромная модель памяти) – практически то же, что и LARGE. Замечания относительно использования пакетов MASM и TASM см. в описании параметра COMPACT;

TCHUGE (огромная модель памяти, совместимая с языком Си) – практически то же, что и HUGE. При использовании для подготовки программы пакета TASM допустимо создание отдельного сегмента стека с помощью директивы .STACK (ср. приведенные в описании параметра COMPACT замечания относительно использования пакетов MASM и TASM);

FLAT (плоская модель памяти) – приложение работает в линейном адресном пространстве объемом 4 Гбайт. Сегменты команд и данных создаются с помощью директив .CODE и .DATA и могут иметь размер до 4 Гбайт. Необходимость в выделении нескольких сегментов команд или данных в этой модели памяти отпадает.

Параметр *язык* может принимать значения, соответствующие ряду языков программирования. Чаще других используются значения PASCAL, C, CPP и STDCALL. Этот параметр определяет интерфейс с вызываемыми процедурами, а также включаемые в каждую процедуру фрагменты входа и выхода. При задании параметра *язык* все процедуры, включенные в программу, если для них в явном виде не указан другой язык, используют интерфейс указанного в директиве .MODEL языка.

.CODE Открытие сегмента команд. Формат директивы:

```
.CODE [имя]
```

Начинает или продолжает сегмент команд. Используется при наличии директивы .MODEL. При использовании малой или компактной модели памяти сегмент команд может быть только один и придание ему имени не допускается. При использовании других моделей памяти допустимо определение нескольких сегментов команд; в этом случае они должны иметь различающиеся имена.

.DATA Открытие сегмента данных. Формат директивы:

```
.DATA
```

Начинает или продолжает сегмент данных. Используется при наличии директивы .MODEL преимущественно в малой или средней модели памяти, в которых допустим только один сегмент данных.

.FARDATA Открытие дальнего сегмента данных. Формат директивы:

```
.FARDATA [имя]
```

Начинает или продолжает сегмент данных. Используется при наличии директивы .MODEL в моделях памяти, допускающих несколько сегментов данных. В этом случае всем сегментам даются различающиеся имена.

.STACK Открытие сегмента стека. Формат директивы:

```
.STACK [размер]
```

Открывает сегмент стека. Параметр *размер* определяет размер в байтах выделяемой под стек области и инициализацию регистра SP этой величиной. При отсутствии параметра размер под стек отводится по умолчанию 1 Кбайт и регистр SP инициализируется значением 400h.

Приложение 3

Команды сопроцессора

F2XM1 Вычисление 2x– 1

Команда не требует операндов; она выполняет вычисления по формуле 2x– 1. Переменная x может принимать значения от 0 до 0,5 и перед выполнением команды должна быть помещена в вершину стека. Результат помещается в вершину стека, причем исходное значение x не сохраняется. Команда влияет на флаги PE, UE, DE.

Пример

```
;Сегмент команд
        fld    x             ;ST=0.1
        f2xml                ;ST=20.1-1=0.07177...
;Сегмент данных
x       dd     0.1
```

FABS Абсолютное значение числа

Команда не требует операндов; она вычисляет абсолютное значение числа, маскируя разряд знака числа из ST. Команда влияет на флаг IE.

Пример

```
;Сегмент команд
        fld    x             ;ST=-1.525e34
        fabs                 ;ST=1.525e34
;Сегмент данных
x       dd     -1.525e34
```

FADD Сложение двух действительных чисел

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда выполняет их сложение и засылает результат в первый операнд. Операндами могут быть только регистры стека:

```
        fadd   ST,ST(i)      ;ST=ST(i)+ST
        fadd   ST(i),ST      ;ST(i)=ST(i)+ST
```

Если указан один операнд, то он суммируется с содержимым вершины стека, результат помещается в вершину стека. Операндом может быть только переменная, описанная как двойное или четверное слово:

```
        fadd   mem           ;ST=ST+mem
```

При отсутствии операндов складывается содержимое вершины стека ST и находящегося под ней регистра ST(1). Результат записывается в регистр ST(1). После этого выполняется операция выталкивания из стека:

```
        fadd                 ;ST(1)=ST(1)+ST,выталкивание
```

Команда влияет на флаги PE, UE, OE, DE, IE.

Пример 1

```
        fadd   ST,ST(5)      ;ST=ST+ST(5)
```

Пример 2

```
;Сегмент команд               ST      ST(1)  ST(2)  ST(3)
        fld    x1            ;4.8      ?      ?      ?
```

```
        fld   x2          ;1.4    4.8     ?       ?
        fld   x3          ;5.2    1.4    4.8      ?
        fadd  ST(2),ST    ;5.2    1.4   10.0      ?
;Сегмент данных
x1      dd    4.8
x2      dd    1.4
x3      dd    5.2
```

Пример 3

```
        fadd  ST,ST       ;ST=ST*2
```

Пример 4

```
;Сегмент команд
        fadd  x1          ;ST=ST+x1
;Сегмент данных
x1      dd    3.2e38      ;Двойное слово
```

Пример 5

```
;Сегмент команд             ST       ST(1)
        fld   a12         ;0.4e308    ?
        fadd  dat         ;1.7e308    ?
;Сегмент данных
a12     dq    0.4e308     ;Четверное слово
dat     dq    1.3e308     ;Четверное слово
```

Пример 6

```
;Сегмент команд           ST      ST(1)  ST(2)
        fld   x1          ;41.8     ?      ?
        fld   x2          ;12.4    41.8    ?
        fadd              ;54.2     ?      ?
;Сегмент данных
x1      dd    41.8
x2      dd    12.4
```

FADDP Сложение двух действительных чисел с записью результата в любой регистр стека и последующим выталкиваиием из стека

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда выполняет их сложение и засылает результат в первый операнд. Затем производится выталкивание из стека. Операндами могут быть только регистры стека. В качестве второго операнда может быть использован только регистр ST:

```
        faddp  ST(i),ST    ;ST(i)=ST(i)+ST, выталкивание
```

Если указан один операнд, то он суммируется с содержимым вершины стека, результат помещается в операнд. Затем производится выталкивание из стека. Операндом может быть любой регистр стека. Эта команда имеет тот же код, что и команда fadd ST(i),ST, она просто является другой формой ее записи:

```
        faddp  ST(i)       ;ST(i)=ST(i)+ST,выталкивание
```

При отсутствии операндов складывается содержимое вершины стека ST и находящейся под ней ячейки ST(1). Результат записывается в регистр ST(1). Затем производится выталкивание из стека. Данная команда полностью аналогична команде fadd без операндов:

```
        faddp              ;ST(1)=ST+ST(1),выталкивание
```

Данная команда аналогична команде faddp ST(3),ST.

Команда влияет на флаги PE, UE, OE, DE, IE.

Пример 1

```
        faddp  ST(4),ST       ;ST(4)=ST(4)+ST, выталкивание
```

Пример 2

```
;Сегмент команд              ST     ST(1)  ST(2)  ST(3)
        fld    x1            ;4.8     ?      ?      ?
        fld    x2            ;1.4    4.8     ?      ?
        fld    x3            ;5.2    1.4    4.8     ?
        faddp  ST(2),ST      ;1.4    10.0    ?      ?
;Сегмент данных
x1      dd     4.8
x2      dd     1.4
x3      dd     5.2
```

Пример 3

```
        faddp  ST(3)          ;ST(3)=ST(3)+ST, выталкивание
```

Пример 4

```
        faddp  ST             ;ST=ST+ST. Результат выталкивается из
                              ;стека и теряется. Команда не имеет смысла
```

FBLD Преобразование операнда из двоично-десятичного кода в формат действительных чисел и загрузка в стек

Команда использует один операнд. Она преобразует упакованный двоично-десятичный операнд в действительный, загружая результат в стек. Знак сохраняется. Команда влияет на флаг IE.

Пример

```
;Сегмент данных
tn      dt 999999999999999999
;Сегмент команд
        fbld   tn             ;ST=999999999999999999
```

FBSTP Преобразование числа из вершины стека в упакованный десятичный формат и выталкивание из стека

Команда использует один операнд. Число, находящееся в вершине стека, округляется. Полученная целая часть преобразуется в упакованный десятичный формат. Результат запоминается в ячейке памяти, адрес которой определяется операндом программы. После этого выполняется выталкивание из стека. Команда влияет на флаг IE.

Пример

```
;Сегмент данных
ddf     dt     ?
ff      dw     99h
;Сегмент команд
        fild   ff             ;ST=153
        fbstp  ddf            ;ST=?, ddf=153
```

FCHS Инвертирование знака числа, находящегося в вершине стека

Команда не требует операндов. Она изменяет знак содержимого вершины стека. Команда влияет на флаг IE.

Пример

```
;Сегмент данных
m1      dd     0.015
;Сегмент команд
        fld    m1             ;ST=0.015
        fchs                  ;ST=-0.015
```

FCLEX Очистка флагов исключительных ситуаций сопроцессора

Команда fclex и аналогичная ей команда fnclex не требуют операндов. Они очищают все флаги исключительных ситуаций, а также поле занятости и поле запроса прерывания в слове состояния сопроцессора. Команда fclex перед очисткой выполняет, а команда fnclex не выполняет проверку условий ошибки с плавающей точкой.

FCOM Сравнение двух действительных чисел

Имеются два варианта использования команды fcom и схожей с ней команды fcomp: без операндов и с одним операндом.

При использовании операнда выполняется его сравнение с содержимым вершины стека. Для этого он вычитается из содержимого вершины стека и по результатам устанавливаются разряды C0, C2 и C3 слова состояния сопроцессора (табл. П-3.1).

Таблица П-3.1. Значения разрядов C0, C2 и C3 слова состояния сопроцессора после выполнения команды сравнения

Результат операции	*Разряды слова состояния*		
	C3	*C2*	*C0*
ST несравнимо с операндом	1	1	1
ST равно операнду	1	0	0
ST меньше операнда	0	0	1
ST больше операнда	0	0	0

Операндом может быть любой регистр стека сопроцессора или переменная, описанная как двойное или четверное слово:

```
fcom   mem          ;ST-mem
fcom   ST(i)        ;ST-ST(i)
```

Если операнды отсутствуют, то сравнивается содержимое ST и ST(1):

```
fcom                ;ST-ST(1)
```

При использовании команды fcomp после сравнения выполняется операция выталкивания из стека:

```
fcomp mem           ;ST-mem, выталкивание
fcomp ST(i)         ;ST-ST(i), выталкивание
fcomp               ;ST-ST(1), выталкивание
```

Команда влияет на флаги DE, IE.

Пример 1

```
;Сегмент данных
dlt     dd     0.5e-8
;Сегмент команд
        fcom   dlt          ;Выполняется операция ST-dlt. По ее
                            ;результату устанавливаются C0, C2
                            ;C3.Содержимое стека не изменяется
```

Пример 2

```
;Сегмент данных
fl      dq     0.5e-100
xx      dd     0.06
;Сегмент команд
        fld    xx           ;ST=0.06
        fcom   fl           ;Выполняется операция ST-fl
                            ;C3=0, C2=0, C0=0; ST=0.06
```

Пример 3

```
;Сегмент данных
xe      dd     100.0
xt      dd     19.8
;Сегмент команд
        fld    xt            ;ST=19.8, ST(1)=?
        fcomp  xe            ;Выполняется операция ST-xe
                             ;C3=0, C2=0, C0=1; ST=?
```

Пример 4

```
;Сегмент данных
xe      dd     100.0
xt      dd     19.8
;Сегмент команд               ST      ST(1)   ST(2)
        fld    xe            ;100.0    ?       ?
        fld    xt            ;19.8    100.0    ?
        fcom                 ;19.8    100.0    ? C3=0, C2=0, C0=1
```

Пример 5

```
;Сегмент данных
xe      dd     100.0
xt      dd     100.0
;Сегмент команд               ST      ST(1)   ST(2)
        fld    xe            ;100.0    ?       ?
        fld    xt            ;100.0   100.0    ?
        fcomp                ;100.0    ?       ? C3=1, C2=0, C0=0;
```

FCOMP Сравнение двух действительных чисел и выполнение операции выталкивания из стека

См. описание команды fcom.

FCOMPP Сравнение двух действительных чисел и выталкивание их из стека

Команда не требует операндов. Она сравнивает содержимое вершины стека ST с содержимым регистра ST(1). Для этого выполняется операция ST-ST(1) и по результатам вычитания устанавливаются разряды C0, C2 и C3 слова состояния сопроцессора (табл. П-3.2). После сравнения два раза выполняется операция чтения из стека, т. е. из стека выталкиваются оба сравниваемых числа:

```
        fcompp               ;Выполняется операция ST-ST(1),
                             ;выталкивание, выталкивание
```

Таблица П-3.2. Значения разрядов C0, C2 и C3 слова состояния сопроцессора после выполнения команды fcompp

Результат операции	*Разряды слова состояния*		
	C3	C2	C0
ST несравнимо с ST(1)	1	1	1
ST равно ST(1)	1	0	0
ST меньше ST(1)	0	0	1
ST больше ST(1)	0	0	0

Команда влияет на флаги DE, IE.

Пример 1

```
;Сегмент данных
xe      dd     100.0
xt      dd     100.0
;Сегмент команд               ST      ST(1)   ST(2)
        fld    xe            ;100.0    ?       ?
```

```
        fld    xt            ;100.0     100.0     ?
        fcompp               ;  ?         ?       ? C3=1, C2=0, C0=0;
```

FCOS Вычисление косинуса (80387+).

Команда не требует операндов. Она вычисляет косинус действительного числа, расположенного в вершине стека, при этом предполагается, что аргумент задан в радианах. Результат выполнения команды помещается в стек на место операнда. Следует иметь в виду, что модуль величины угла не должен превышать значения 263, причем следить за этим должен сам программист. Если значение аргумента выходит из заданного диапазона, то после выполнения команды ST не изменится и установится флаг C2 слова состояния. Команда влияет на флаги PE, DE, IE.

Пример

```
                             ;ST          ST(1)
        fldpi                ;3.1416...    ?
        fcos                 ;-1.0         ?
```

FDECSTP Уменьшение указателя стека

Команда не требует операндов. После ее выполнения указатель стека уменьшается на единицу, т. е. все регистры проталкиваются в стек. При этом в вершину стека заносится содержимое регистра ST(7). Команда не влияет на флаги.

Пример

```
        fdecstp
```

Результат выполнения команды проиллюстрирован на рис. П-3.1.

Содержимое стека

	До выполнения команды	После выполнения команды
ST	0	7
ST(1)	1	0
ST(2)	2	1
ST(3)	3	2
ST(4)	4	3
ST(5)	5	4
ST(6)	6	5
ST(7)	7	6

Рис. П-3.1. Результат выполнения команды fdecstp

FDIV Деление двух действительных чисел

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда делит содержимое первого операнда на содержимое второго и засылает результат в первый операнд. Операндами могут быть только регистры стека:

```
        fdiv   ST,ST(i)      ;ST=ST/ST(i)
        fdiv   ST(i),ST      ;ST(i)=ST(i)/ST
```

Если указан один операнд, то на него делится содержимое вершины стека и результат помещается в вершину стека. Операндом может быть только переменная, описанная как двойное либо четвертное слово:

```
        fdiv   mem           ;ST=ST/mem
```

При отсутствии операндов содержимое регистра ST(1) делится на содержимое вершины стека ST. Результат записывается в регистр ST(1) и затем выполняется операция выталкивания из стека.

```
        fdiv                 ;ST(1)=ST(1)/ST, выталкивание
```

Команда влияет на флаги PE, UE, OE, ZE, DE, IE.

Пример 1

```
;Сегмент команд               ST     ST(1)  ST(2)  ST(3)
         fld    x1          ; 4.0     ?      ?      ?
         fld    x2          ; 6.3    4.0     ?      ?
         fld    x3          ;10.0    6.3    4.0     ?
         fdiv   ST,ST(2)    ; 2.5    6.3    4.0     ?
;Сегмент данных
x1       dd     4.0
x2       dd     6.3
x3       dd     10.0
```

Пример 2

```
;Сегмент команд               ST     ST(1)  ST(2)  ST(3)
         fld    x1          ; 4.0     ?      ?      ?
         fld    x2          ; 6.3    4.0     ?      ?
         fld    x3          ;10.0    6.3    4.0     ?
         fdiv   ST(2),ST    ;10.0    6.3    0.4     ?
;Сегмент данных
x1       dd     4.0
x2       dd     6.3
x3       dd     10.0
```

Пример 3

```
;Сегмент команд               ST     ST(1)  ST(2)
         fld    x1          ;11.5      ?      ?
         fld    x2          ; 6.3    11.5     ?
         fdiv   ST,ST       ; 1.0    11.5     ?
;Сегмент данных
x1       dd     11.5
x2       dd     6.3
```

Пример 4

```
;Сегмент команд               ST      ST(1)
         fld    a1          ;4.0e307  ?
         fdiv   dlm         ;20.0      ?
;Сегмент данных
a1       dq     0.4e308     ;Четверное слово
dlm      dq     2.0e306     ;Четверное слово
```

Пример 5

```
;Сегмент команд               ST     ST(1)  ST(2)  ST(3)
         fld    x1          ;6.0      ?      ?      ?
         fld    x2          ;2.0     6.0     ?      ?
         fld    x3          ;4.0     2.0    6.0     ?
         fdiv               ;0.5     6.0     ?      ?
;Сегмент данных
x1       dd     6.0
x2       dd     2.0
x3       dd     4.0
```

FDIVP Деление двух действительных чисел с последующим выталкиванием из стека

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда делит содержимое первого операнда на содержимое второго и засылает результат в первый операнд. Затем производится выталкивание из стека. Операндами могут быть только регистры стека. В качестве первого операнда запрещается использовать регистр ST:

```
         fdivp  ST(i),ST    ;ST(i)=ST(i)/ST, выталкивание
```

Если указан один операнд, то он делится на содержимое вершины стека, результат помещается в этот операнд. Затем производится выталкивание из стека. Операндом может быть только регистр стека. Допускается использовать регистр ST:

```
        fdivp  ST(i)        ;ST(i)=ST(i)/ST, выталкивание
```

При отсутствии операндов содержимое регистра ST(1) делится на содержимое вершины стека ST. Результат записывается в регистр ST(1). Затем производится выталкивание из стека:

```
        fdivp               ;ST(1)=ST(1)/ST, выталкивание
```

Данная команда аналогична команде fdiv.

Команда влияет на флаги PE, UE, OE, ZE, DE, IE.

Пример 1

```
;Сегмент команд                ST    ST(1)  ST(2)  ST(3)
        fld    x1             ; 4.0    ?      ?      ?
        fld    x2             ; 1.5   4.0     ?      ?
        fld    x3             ;10.0   1.5    4.0     ?
        fdivp  ST(2),ST       ; 1.5   0.4     ?      ?
;Сегмент данных
x1      dd     4.0
x2      dd     1.5
x3      dd     10.0
```

Пример 2

```
;Сегмент команд                ST    ST(1)  ST(2)
        fld    f              ; 4.0    ?      ?
        fld    s              ;10.0   4.0     ?
        fdivp  ST(1)          ; 0.4           ?
;Сегмент данных
f       dd     4.0
s       dd     10.0
```

Пример 3

```
;Сегмент команд               ST    ST(1)  ST(2)  ST(3)
        fld    x1             ;6.0    ?      ?      ?
        fld    x2             ;2.0   6.0     ?      ?
        fld    x3             ;4.0   2.0    6.0     ?
        fdivp                 ;0.5   6.0     ?      ?
;Сегмент данных
x1      dd     6.0
x2      dd     2.0
x3      dd     4.0
```

FDIVR Деление двух действительных чисел в обратном порядке

Данная команда идентична команде fdiv, но делимос и делитель меняются местами. Варианты использования операндов показаны ниже:

```
        fdivr  ST,ST(i)       ;ST=ST(i)/ST
        fdivr  ST(i),ST       ;ST(i)=ST/ST(i)
        fdivr  mem            ;ST=mem/ST
        fdivr                 ;ST(1)=ST/ST(1), выталкивание
```

Команда влияет на флаги PE, UE, OE, ZE, DE, IE.

Пример 1

```
;Сегмент команд                ST    ST(1)  ST(2)  ST(3)
        fld    x1             ; 4.0    ?      ?      ?
        fld    x2             ; 6.3   4.0     ?      ?
        fld    x3             ;10.0   6.3    4.0     ?
        fdivr  ST,ST(2)       ; 0.4   6.3    4.0     ?
```

```
;Сегмент данных
x1      dd     4.0
x2      dd     6.3
x3      dd     10.0
```

Пример 2

```
;Сегмент команд              ST     ST(1)  ST(2)  ST(3)
        fld    x1            ; 4.0    ?      ?      ?
        fld    x2            ; 6.3   4.0     ?      ?
        fld    x3            ;10.0   6.3    4.0     ?
        fdivr  ST(2),ST      ;10.0   6.3    2.5     ?
;Сегмент данных
x1      dd     4.0
x2      dd     6.3
x3      dd     10.0
```

Пример 3

```
;Сегмент команд              ST         ST(1)
        fld    eps           ;2.6e-38     ?
        fdivr  del           ;19999.9...  ?
;Сегмент данных
eps     dq     2.6e-38       ;Двойное слово
del     dq     5.2e-34       ;Двойное слово
```

Пример 4

```
;Сегмент команд              ST     ST(1)  ST(2)  ST(3)
        fld    x1            ;6.0     ?      ?      ?
        fld    x2            ;2.0    6.0     ?      ?
        fld    x3            ;4.0    2.0    6.0     ?
        fdivr                ;2.0    6.0     ?      ?
;Сегмент данных
x1      dd     6.0
x2      dd     2.0
x3      dd     4.0
```

FDIVRP Деление двух действительных чисел в обратном порядке с последующим выталкиванием из стека

Команда идентична fdivp, но операнды (делимое и делитель) меняются местами. Варианты использования операндов показаны ниже:

```
        fdivrp ST(i),ST      ;ST(i)=ST/ST(i), выталкивание
        fdivrp ST(i)         ;ST(i)=ST/ST(i), выталкивание
        fdivrp               ;ST(1)=ST/ST(1), выталкивание
                             ;Команда аналогична команде fdivr
```

Команда влияет на флаги PE, UE, OE, ZE, DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)     ST(2)  ST(3)
        fld    x1            ;6.0     ?         ?      ?
        fld    x2            ;2.0    6.0        ?      ?
        fld    x3            ;4.0    2.0       6.0     ?
        fdivrp ST(2)         ;2.0    0.666...   ?      ?
;Сегмент данных
x1      dd     6.0
x2      dd     2.0
x3      dd     4.0
```

Пример 2

```
;Сегмент команд              ST     ST(1)  ST(2)  ST(3)
        fld    x1            ;6.0     ?      ?      ?
        fld    x2            ;3.7    6.0     ?      ?
```

```
        fld    x3              ;4.0      3.7      6.0      ?
        fdivrp                 ;1.08... 6.0...    ?        ?
;Сегмент данных
x1      dd     6.0
x2      dd     3.7
x3      dd     4.0
```

FFREE Освобождение регистра стека

Данная команда помещает в тег регистра, который является ее операндом, число 11b. После этого данный регистр рассматривается как пустой и в него можно снова записывать информацию (попытка записи в непустой регистр вызывает исключительную ситуацию):

```
        ffree  ST(i)           ;Регистр ST(i) очищен
```

Команда не влияет на флаги.

FIADD Сложение целого числа с действительным

При использовании этой команды необходимо указывать один целочисленный операнд. Он суммируется с содержимым вершины стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово:

```
        fiadd  mem             ;ST=ST+mem
```

Команда влияет на флаги PE, OE, DE, IE.

Пример

```
;Сегмент команд                  ST           ST(1)
        fld    op              ;0.6785e6       ?
        fadd   dat             ;200000.0       ?
;Сегмент данных
op      dd     0.16785e6       ;Двойное слово
dat     dw     32150           ;Слово
```

FICOM Сравнение заданного целого числа с вещественным числом из вершины стека

При использовании данной команды необходимо указывать один целочисленный операнд. Он преобразуется в действительное представление и сравнивается с содержимым вершины стека. Операндом может быть только переменная, описанная как слово или двойное слово:

```
        ficom  mem             ;ST-mem
```

Данная команда идентична команде fcom.

Команда влияет на флаги DE, IE.

Пример

```
;Сегмент данных
nmb     dd     60000           ;Двойное слово
lim     dd     60000           ;Двойное слово
;Сегмент команд
        fild   lim             ;ST=60000.0, ST(1)=?
        ficom  nmb             ;Выполняется операция ST-nmb
                               ;C3=1, C2=0, C0=0; ST=60000.0,ST(1)=?
```

FICOMP Сравнение заданного целого числа с вещественным числом из вершины стека с последующим выталкиванием из стека

При использовании данной команды необходимо указывать один целочисленный операнд. Он преобразуется в действительное представление и сравнивается с содер-

жимым вершины стека. После этого выполняется операция выталкивания из стека. Операндом может быть только переменная, описанная как слово или двойное слово:

```
        ficomp mem          ;ST-mem, выталкивание
```

Данная команда идентична команде fcomp.

Команда влияет на флаги DE, IE.

Пример

```
;Сегмент данных
nmb     dw    1200          ;Слово
lim     dd    60000         ;Двойное слово
;Сегмент команд
        fild  lim           ;ST=60000.0, ST(1)=?
        ficomp nmb          ;Выполняется операция ST-nmb
                            ;C3=0, C2=0, C0=0; ST=?, ST(1)=?
```

FIDIV Деление действительного числа на заданный целочисленный операнд

При использовании этой команды необходимо указывать один целочисленный операнд. Команда fidiv делит содержимое вершины стека на этот операнд. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово.

```
        fidiv mem           ;ST=ST/mem
```

Команда влияет на флаги PE, UE, OE, ZE, DE, IE.

Пример

```
;Сегмент команд              ST        ST(1)
        fldpi               ;3.141...    ?
        fidiv del           ;0.785...    ?
;Сегмент данных
del     dw    4             ;Слово
```

FIDIVR Деление заданного целочисленного операнда на действительное число

При использовании этой команды необходимо указывать один целочисленный операнд. Команда fidivr делит операнд на действительное число, записанное в вершине стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово:

```
        fidivr mem          ;ST=mem/ST
```

Команда влияет на флаги PE, UE, OE, ZE, DE, IE.

Пример

```
;Сегмент команд              ST        ST(1)
        fldpi               ;3.141...    ?
        fidivr grad         ;57.29...    ?
;Сегмент данных
grad    dd    180           ;Двойное слово
```

FILD Преобразование заданного целочисленного операнда в действительное представление и загрузка его в стек

При использовании этой команды необходимо указывать один целочисленный операнд. Данная команда загружает в стек заданное целое число, предварительно преобразуя его в действительный формат. Операндом может быть только переменная, описанная как слово, двойное слово или четверное слово:

```
        fild  mem           ;Загрузка стека, ST=mem
```

Команда влияет на флаг IE.

Пример

```
;Сегмент команд                 ST         ST(1)   ST(3)
        fild  d1           ;  -126            ?       ?
        fild  d2           ;  56665        -126       ?
        fild  d3           ;6576575611  56665    -126
;Сегмент данных
d1      dw    -126         ;Слово
d2      dd    56665        ;Двойное слово
d3      dq    6576575611;Четверное слово
```

FIMUL Умножение заданного целого числа на действительное

Команда требует одного целочисленного операнда, который умножается на содержимое вершины стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово.

```
        fimul  mem         ;ST=ST*mem
```

Команда влияет на флаги PE, OE, DE, IE.

Пример

```
;Сегмент команд               ST         ST(1)
        fldpi              ;3.141...     ?
        fimul  kf          ;6.283...     ?
;Сегмент данных
kf      dw    2            ;Слово
```

FINCSTP Увеличение указателя стека

Команда не требует операндов. После ее выполнения указатель стека увеличивается на единицу, т. е. выполняется операция выталкивания из стека. Таким образом, в вершине стека будет находиться содержимое регистра ST(1). Содержимое старой вершины стека заносится в регистр ST(7). Команда не влияет на флаги. Результат выполнения команды проиллюстрирован на рис. П-3.2.

Содержимое стека

	До выполнения команды	После выполнения команды
ST	0	1
ST(1)	1	2
ST(2)	2	3
ST(3)	3	4
ST(4)	4	5
ST(5)	5	6
ST(6)	6	7
ST(7)	7	0

Рис. П-3.2. Результат выполнения команды fincstp

FINIT Инициализация сопроцессора

Команда finit и аналогичная ей команда fninit не требуют операндов. Их действие подобно аппаратному сбросу сопроцессора. После выполнения этих команд слово управления устанавливается в 037Fh (округлять к ближайшему, все особые ситуации замаскированы, точность 64 бита), слово состояния очищается, а в слове признаков устанавливается 0FFFFh (все регистры пустые). Команда finit перед выполнением инициализации проверяет наличие немаскируемых условий ошибки для операций с плавающей точкой, а команда fninit этого не делает. Команды не влияют на флаги.

FIST Преобразование числа из вершины стека в целое число и запись его в поле памяти

При использовании этой команды необходимо указывать один операнд. Данная команда округляет число из вершины стека до целого значения и записывает его в указанный операнд, которым может быть только переменная, описанная как слово, двой-

ное слово или длинное целое, занимающее 4 слова. Способ округления определяется содержимым поля RC регистра управления сопроцессора. Если RC=00b, то выполняется округление до ближайшего целого. Если RC=01b или 10b, то число округляется в сторону уменьшения или увеличения соответственно. При RC=11b происходит отбрасывание дробной части:

```
        fist   mem           ;mem=ST
```

Команда влияет на флаги PE, IE.

Пример 1

```
;Содержимое поля RC регистра управления равно 00b
;Сегмент команд                  ST        ST(1)     xx
                             ;12.459...     ?         ?
        fist   xx            ;12.459...     ?         12
;Сегмент данных
xx      dw     ?             ;Слово
```

Пример 2

```
;Содержимое поля RC регистра управления равно 10b
;Сегмент команд                  ST        ST(1)     xx
                             ;12.459...     ?         ?
        fist   xx            ;12.459...     ?         13
;Сегмент данных
xx      dw     ?             ;Слово
```

FISTP Преобразование числа из вершины стека в целое число, запись его в поле памяти и выполнение операции выталкивания из стека

При использовании этой команды необходимо указывать один операнд. Данная команда округляет число из вершины стека до целого значения и записывает его в указанный операнд, которым может быть только переменная, описанная как слово, двойное слово или длинное целое, занимающее 4 слова. Способ округления определяется содержимым поля RC регистра управления сопроцессора. Если RC=00b, то выполняется округление до ближайшего целого. Если RC=01b или 10b, то число округляется в сторону уменьшения или увеличения соответственно. При RC=11b происходит отбрасывание дробной части числа. После записи результата в память выполняется операция выталкивания из стека:

```
        fistp  mem           ;mem=ST, выталкивание
```

Команда влияет на флаги PE, IE.

Пример 1

```
;Содержимое поля RC регистра управления равно 00b
;Сегмент команд                 ST         ST(1)     xx
                             ;2.59...       ?         ?
        fistp  xx            ;   ?          ?         2
;Сегмент данных
xx      dw     ?             ;Слово
```

Пример 2

```
;Содержимое поля RC регистра управления равно 11b
;Сегмент команд                  ST        ST(1)     xx
                             ;2.59...       ?         ?
        fistp  xx            ;   ?          ?         2
;Сегмент данных
xx      dw     ?             ;Слово
```

FISUB Вычитание заданного целого числа из действительного

При использовании этой команды необходимо указывать один целочисленный операнд. Он вычитается из содержимого вершины стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово:

```
        fisub  mem          ;ST=ST-mem
```

Команда влияет на флаги PE, OE, DE, IE.

Пример

```
;Сегмент команд                  ST    ST(1)
                                ;125.14   ?
        fisub  g3               ;  5.14   ?
;Сегмент данных
g3      dw     120
```

FISUBR Вычитание действительного числа из заданного целого

При использовании этой команды необходимо указывать один целочисленный операнд. Из него вычитается содержимое вершины стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как слово или двойное слово:

```
        fisubr mem          ;ST=mem-ST
```

Команда влияет на флаги PE, OE, DE, IE.

Пример

```
;Сегмент команд                  ST    ST(1)
                                ;125.14   ?
        fisubr g3               ; -5.14   ?
;Сегмент данных
g3      dw     120
```

FLD Загрузка в стек действительного числа

При применении этой команды необходимо указывать один операнд. Команда загружает в стек действительное число, преобразуя его в формат, используемый в сопроцессоре. Операндом может быть регистр стека или переменная, занимающая двойное, четверное или десятерное слово:

```
fld     ST(i)  ;Содержимое ST(i) загружается в ST
fld     mem    ;Содержимое mem загружается в ST
```

Попытка выполнения команды fld при полностью загруженном стеке приводит к тому, что выполнятся операция записи, однако содержимое вершины стека становится неопределенным (рис. П-3.3). Команда влияет на флаг IE.

Содержимое стека

	До выполнения команды	После выполнения команды
ST	0	NAN
ST(1)	1	0
ST(2)	2	1
ST(3)	3	2
ST(4)	4	3
ST(5)	5	4
ST(6)	6	5
ST(7)	7	6

Рис. П-3.3. Результат выполнения команды fld при полностью загруженном стеке (NAN – значение регистра не определено)

Пример

```
;Сегмент команд                ST          ST(1)       ST(2)     ST(3)
        fld    e34         ;3.2e38        ?            ?         ?
        fld    m2          ;1.8e-108    3.2e38         ?         ?
        fld    mt          ;1.05e4932  1.8e-108     3.2e38       ?
        fld    ST(2)       ;3.2e38     1.05e4932   1.8e-108   3.2e38
;Сегмент данных
m2      dq     1.8e-108    ;Четверное слово
e34     dd     3.2e38      ;Двойное слово
mt      dt     1.05e4932;Десятерное слово
```

FLDCW Загрузка слова управления сопроцессора

Команда использует один операнд, который загружается в регистр управления. Предварительно слово управления должно быть сохранено с помощью команд fstcw или fnstcw. Если разряды исключительных ситуаций установлены до выполнения команды fldcw и новое слово не маскирует исключительные ситуации, то удовлетворяется немедленный запрос следующей команды. Команда не влияет на флаги.

Пример

```
;Сегмент данных
singl   dw     ?           ;Поле памяти для хранения слова управления
;Сегмент команд
        fldcw  singl
```

FLDENV Загрузка рабочей среды сопроцессора

Команда использует один операнд. В результате ее выполнения загружается рабочая среда сопроцессора из 14- или 28-байтового буфера памяти, заданного операндом. Эта информация должна быть предварительно записана в буфер с помощью команды fstenv. Размер операнда определяется тем, какой атрибут use установлен для текущего сегмента кодов. Если установлен атрибут use16, то используется 14-байтовый буфер, в случае use32 – 28-байтовый буфер. Если разряды исключительных ситуаций установлены до выполнения команды и новое слово не маскирует исключительные ситуации, то удовлетворяется немедленный запрос следующей команды. Команда не влияет на флаги.

Примеры

```
;Сегмент данных
        buf    db          14 dup (?)
;Сегмент команд
        code segment use16
        ...
        fstenv buf
        ...
        fldenv buf
```

FLD1, FLDL2T, FLDL2E, FLDPI, FLDLG2, FLDLN2, FLDZ Загрузка в стек предопределенных констант

Каждая из этих команд предназначена для загрузки в стек определенной константы. Точность всех констант, за исключением 0 и 1, составляет 19 десятичных разрядов:

```
        fld1               ;Загрузка в стек единицы
        fldl2t             ;Загрузка в стек логарифма по
                           ;основанию 2 числа 10
        fldl2e             ;Загрузка в стек логарифма по
                           ;основанию 2 числа e
```

```
        fldpi           ;Загрузка в стек числа pi
        fldlg2          ;Загрузка в стек логарифма по
                        ;основанию 10 числа 2
        fldln2          ;Загрузка в стек логарифма по
                        ;основанию e числа 2
        fldz            ;Загрузка в стек нуля
```

Команда влияет на флаг IE.

Пример

```
                        ;ST       ST(1)    ST(2)    ST(3)   ST(4)
        fldz            ;0.0       ?        ?        ?       ?
        fld1            ;1.0      0.0       ?        ?       ?
        fldpi           ;3.14...  1.0      0.0       ?       ?
        fldl2t          ;3.32...  3.14...  1.0      0.0      ?
        fldlg2          ;0.30...  3.32...  3.14...  1.0     0.0
```

FMUL Умножение двух действительных чисел

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда умножает содержимое второго операнда на содержимое первого и засылает результат в первый операнд. Операндами могут быть только регистры стека:

```
        fmul   ST,ST(i)     ;ST=ST*ST(i)
        fmul   ST(i),ST     ;ST(i)=ST(i)*ST
```

Если указан один операнд, то он умножается на содержимое вершины стека. Полученный результат помещается в вершину стека. Операндом может быть только переменная, описанная как двойное или четверное слово:

```
        fmul   mem          ;ST=ST*mem
```

При отсутствии операндов содержимое вершины стека ST умножается на содержимое находящейся под ней ячейки ST(1). После этого выполняется выталкивание из стека и результат записывается в вершину стека:

```
        fmul                ;ST(1)=ST(1)*ST, выталкивание
```

Команда влияет на флаги PE, UE, OE ,DE, IE.

Пример 1

```
;Сегмент команд              ST    ST(1)   ST(2)   ST(3)
        fld    x1           ;4.0    ?       ?       ?
        fld    x2           ;6.3   4.0      ?       ?
        fld    x3           ;0.4   6.3     4.0      ?
        fmul   ST,ST(2)     ;1.6   6.3     4.0      ?
;Сегмент данных
x1      dd     4.0
x2      dd     6.3
x3      dd     0.4
```

Пример 2

```
;Сегмент команд              ST    ST(1)   ST(2)   ST(3)
        fld    x1           ;4.0    ?       ?       ?
        fld    x2           ;6.3   4.0      ?       ?
        fld    x3           ;0.4   6.3     4.0      ?
        fmul   ST(2),ST     ;0.4   6.3     1.6      ?
;Сегмент данных
x1      dd     4.0
x2      dd     6.3
x3      dd     0.4
```

Пример 3

```
;Сегмент команд              ST      ST(1)
        fld     a1          ;4.0e26  ?
        fmul    s3          ;0.08    ?
;Сегмент данных
a1      dq      0.4e26      ;Четверное слово
s3      dq      2.0e-28     ;Четвернсе слово
```

Пример 4

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ;6.0     ?       ?       ?
        fld     x2          ;2.5    6.0      ?       ?
        fld     x3          ;2.0    2.5     6.0      ?
        fmul                ;5.0    6.0      ?       ?
;Сегмент данных
x1      dd      6.0
x2      dd      2.5
x3      dd      2.0
```

FMULP Умножение двух действительных чисел с последующим выталкиванием из стека

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда умножает содержимое второго операнда на содержимое первого и засылает результат в первый операнд. Затем производится выталкивание из стека. Операндами могут быть только регистры стека. В качестве первого операнда запрещается использовать регистр ST:

```
        fmulp  ST(i),ST     ;ST(i)=ST(i)*ST, выталкивание
```

Если указан один операнд, то он умножается на содержимое вершины стека, результат помещается в этот операнд. Затем производится выталкивание из стека. Операндом может быть только регистр стека. Допускается использовать регистр ST:

```
        fmulp  ST(i)        ;ST(i)=ST(i)*ST, выталкивание
```

При отсутствии операндов содержимое вершины стека ST умножается на содержимое находящейся под ней ячейки ST(1). Результат записывается в регистр ST(1). Затем производится выталкивание из стека:

```
        fmulp               ;ST(1)=ST(1)*ST, выталкивание
```

Эта команда аналогична команде fmul.

Команда влияет на флаги PE, UE, OE, ZE, DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld     x1          ;4.0     ?       ?       ?
        fld     x2          ;0.7    4.0      ?       ?
        fld     x3          ;0.4    0.7     4.0      ?
        fmulp   ST(2)       ;0.7    1.6      ?       ?
;Сегмент данных
x1      dd      4.0
x2      dd      0.7
x3      dd      0.4
```

Пример 2

```
;Сегмент команд              ST     ST(1)   ST(2)
        fld     x1          ;4.0     ?       ?
        fld     x2          ;0.7    4.0      ?
        fmulp   ST          ;4.0     ?       ?
```

```
;Сегмент данных
x1       dd     4.0
x2       dd     0.7
```

Пример 3

```
;Сегмент команд              ST      ST(1)   ST(2)   ST(3)
         fld    x1           ;6.0     ?       ?       ?
         fld    x2           ;2.5     6.0     ?       ?
         fld    x3           ;2.0     2.5     6.0     ?
         fmulp               ;5.0     6.0     ?       ?
;Сегмент данных
x1       dd     6.0
x2       dd     2.5
x3       dd     2.0
```

FNCLEX Очистка флагов исключительных ситуаций сопроцессора

См. описание команды fclex.

FNINIT Инициализация сопроцессора

См. описание команды finit.

FNOP Нет операции

Команда nop не выполняет никакой операции. Она влияет только на значение указателя команд. Команда не влияет на флаги.

FNSAVE Запись состояния сопроцессора

См. описание команды fsave.

FNSTCW Запись слова управления сопроцессора

См. описание команды fstcw.

FNSTENV Запись рабочей среды сопроцессора

См. описание команды fstenv.

FNSTSW Запись слова состояния сопроцессора

См. описание команды fstsv.

FPATAN Вычисление арктангенса угла

Команда не требует операндов. Она вычисляет арктангенс величины, равной ST(1)/ST. При выполнении этой команды производится выталкивание из стека и, таким образом, полученное значение угла замещает оба операнда. Значение угла вычисляется в радианах:

```
         fpatan              ;ST(1)=arctg[ST(1)/ST], выталкивание
```

Команда влияет на флаги PE, UE, OE, DE.

Пример

```
;Сегмент команд              ST        ST(1)     ST(2)
         fldl                ;1.0       ?         ?
         fldl                ;1.0       1.0       ?
         fpatan              ;0.78...   ?         ?
         fldpi               ;3.14...   0.78...   ?
         fdiv                ;0.25...   ?         ?
         fimul  c            ;45.0      ?         ?
;Сегмент данных
c        dw     180
```

FPREM Вычисление частичного остатка деления

Команда не требует операндов. Она вычисляет остаток, полученный от деления ST на ST(1). Знак остатка тот же, что и знак исходного делимого в ST. Команда влияет на флаги UE, DE, IE.

Пример

```
;Сегмент команд                ST      ST(1)     ST(2)
         fld    op1            ;10.0     ?          ?
         fld    op2            ;17.0    10.0        ?
         fprem                 ;7.0...  10.0        ?
;Сегмент данных
op1      dd     10.0
op2      dd     17.0
```

FPREM1 Вычисление частичного остатка деления по стандарту IEEE-754 (80387+)

Команда не требует операндов. Она вычисляет остаток, полученный от деления ST на ST(1), и оставляет остаток в ST. По этой команде выполнястся операция ST=ST–ST(1), до тех пор, пока не выполнится условие ST<ST(1)/2. Команда влияет на флаги UE, DE, IE.

Пример 1

```
;Сегмент команд                ST      ST(1)     ST(2)
         fld    op1            ;10.0     ?          ?
         fld    op2            ;17.0    10.0        ?
         fprem                 ;-3.0...  10.0        ?
;Сегмент данных
op1      dd     10.0
op2      dd     17.0
```

Пример 2

```
;Сегмент команд                ST          ST(1)     ST(2)
         fld    op1            ;10.0         ?          ?
         fld    op2            ;14.9999     10.0        ?
         fprem                 ; 4.9999     10.0        ?
;Сегмент данных
op1      dd     10.0
op2      dd     14.9999
```

FPTAN Вычисление тангенса угла

Команда не требует операндов. Она вычисляет тангенс угла, заданного в вершине стека. Угол должен быть задан в радианах.

Выполнение этой команды зависит от типа сопроцессора. При использовании сопроцессоров 8087 и 80287 значение угла должно находиться в диапазоне от 0 до ?/4. Результат в виде двух величин X и Y записывается в ST(1) и ST. Для завершения вычисления надо разделить ST на ST(1). При использовании сопроцессоров 80387+ значение модуля угла не должно превышать 263. После выполнения команды угол заменяется его тангенсом, вслед за чем в стек помещается число 1.0.

Команда влияет на флаги PE, DE, IE.

Пример (для 80387+)

```
;Сегмент команд                ST          ST(1)
         fldpi                 ;3.14...      ?
         fidiv  c              ;0.785...     ?
         fptan                 ;1           1.732...
;Сегмент данных
c        dw     3
```

FRNDINT Округление действительного числа из вершины стека до целого значения

Команда не требует операндов. Она округляет действительное число из вершины стека до целого в соответствии со значением поля RC слова управления сопроцессора. Результат заменяет исходное действительное число в вершине стека. Если RC=00b, то выполняется округление до ближайшего целого. Если RC=01b или 10b, то число округляется в сторону уменьшения или увеличения соответственно. При RC=11b происходит отбрасывание дробной части числа. Для того чтобы изменить режим округления, можно записать слово управления командой fstcw и, изменив содержимое поля RC, загрузить его снова с помощью команды fldcw. Команда влияет на флаги PE, IE.

Пример

```
;Сегмент данных
nmb      dd     23.6666667
;Сегмент команд
         fld    nmb             ;ST<-nmb
         frndint                ;ST=24, при RC=00b или 10b
                                ;ST=23, при RC=01b или 11b
```

FRSTOR Восстановление состояния сопроцессора

Команда использует один операнд. Она восстанавливает полное состояние сопроцессора по содержимому буфера памяти (размером 94 или 108 байт), заданному этим операндом. Размер буфера определяется тем, какой атрибут use установлен для текущего сегмента кодов. Если установлен атрибут use16, то используется 94-байтовый буфер, а если установлен атрибут use32, то 108-байтовый буфер. Для того чтобы сформировать содержимое буфера, предварительно используется команда fsave. Команда не влияет на флаги.

Пример

```
;Сегмент данных
buf      db     94 dup (?)
;Сегмент команд
code     segment use16
         ...
         fsave  buf
         ...
         frstor buf
```

FSAVE Запись состояния сопроцессора

Команда fsave и аналогичная ей команда fnsave используют один операнд. Они записывают полное состояние сопроцессора в буфер памяти размером 94 байта или 108 байт, заданный этим операндом. Размер буфера определяется тем, какой атрибут use установлен для текущего сегмента кодов. Если установлен атрибут use16, то используется 94-байтовый буфер, а если установлен атрибут use32, то 108-байтовый буфер. После выполнения этих команд производится инициализация сопроцессора. При выполнении команды fsave перед записью состояния производится проверка условия немаскируемой ошибки с плавающей точкой. При выполнении команды fnsave такая проверка не производится. Команды не влияют на флаги.

Примеры

```
;Сегмент данных
buf      db     94 dup (?)
;Сегмент команд
code     segment use16
         ...
         fnsave buf
```

FSCALE Масштабирование действительного числа

Команда не требует операндов. Она умножает содержимое вершины стека ST на 2 в степени ST(1). Если содержимое регистра ST(1) оказывается нецелым, команда использует ближайшее меньшее по величине целое число. Таким образом, с помощью этой команды можно выполнять умножение или деление на целочисленные степени двух. Команда влияет на флаги UE, OE, IE.

Пример 1

```
;Сегмент команд                ST           ST(1)
        fild   pow            ;5.0            ?
        fild   p1             ;8.0           5.0
        fscale                ;256.0         5.0
;Сегмент данных
p1      dw     8
pow     dw     5
```

Пример 2

```
;Сегмент команд                ST           ST(1)
        fild   pow            ;2.0            ?
        fild   p1             ;73.87         2.0
        fscale                ;295.48        2.0
;Сегмент данных
p1      dd     73.87
pow     dw     2
```

FSIN Вычисление синуса (80387+)

Команда не требует операндов. Она вычисляет синус действительного числа, расположенного в вершине стека, при этом предполагается, что аргумент задан в радианах. Результат выполнения команды помещается в стек на место операнда. Следует иметь в виду, что модуль величины угла не должен превышать двух в степени 63, причем следить за этим должен сам программист. Если значение аргумента выходит из заданного диапазона, то после выполнения команды ST не изменится и будет установлен флаг C2 слова состояния:

```
        fsin                  ;ST=sin(ST)
```

Команда влияет на флаги PE, DE, IE.

Пример

```
                              ;ST          ST(1)
        fldpi                 ;3.1416...    ?
        fsin                  ;0.0          ?
```

FSINCOS Одновременное вычисление синуса и косинуса (80387+)

Команда не требует операндов. Она вычисляет синус и косинус действительного числа, расположенного в вершине стека, при этом предполагается, что аргумент задан в радианах. Результат выполнения команды помещается в стек на место операнда. Следует иметь в виду, что модуль величины угла не должен превышать двух в степени 63, причем следить за этим должен программист. Если значение аргумента выходит из заданного диапазона, то после выполнения команды ST не изменится и будет установлен флаг C2 слова состояния. После вычисления значение косинуса помещается в регистр ST, а значение синуса – в ST(1):

```
        fsincos               ;ST=cos(ST), ST(1)=sin(ST)
```

Команда влияет на флаги PE, DE, IE.

Пример

```
                                ;ST           ST(1)
        fldpi                   ;3.1416...     ?
        fsincos                 ;-1.0         0.0
```

FSQRT Вычисление квадратного корня

Команда не требует операндов. Она вычисляет квадратный корень положительного действительного числа, расположенного в вершине стека. Результат помещается на место исходного числа:

```
        fsqrt                   ;ST=sqrt(ST)
```

Команда влияет на флаги PE, DE, IE.

Пример

```
;Сегмент команд                  ST           ST(1)
        fild   p1               ;2.0            ?
        fsqrt                   ;1.414...       ?
;Сегмент данных
p1      dw     2
```

FST Запись действительного числа

При использовании этой команды необходимо указывать один операнд, в который копируется содержимое вершины стека. Операндом может быть регистр стека или переменная, занимающая двойное или четверное слово. Перед записью выполняется округление в соответствии со значением поля RC управляющего слова до размеров операнда. Если RC=00b, то выполняется округление до ближайшего числа. Если RC=01b или 10b, то число округляется в сторону уменьшения или увеличения соответственно. При RC=11b лишние цифры дробной части числа отбрасываются:

```
        fst    mem              ;mem=ST
        fst    ST(i)            ;ST(i)=ST
```

Команда влияет на флаги PE, UE, OE, IE.

Пример 1

```
                ;   ST           ST(1)    ST(2)
                                ;1.2459e1976     ?         ?
        fst    ST(2)            ;1.2459e1976     ?   1.2459e1976
```

Пример 2

```
;Сегмент команд                 ;   ST            ST(1)   zo
                                ;1.34e-189...      ?       ?
        fst    zo               ;1.34e-189...      ?       0
;Сегмент данных
zo      dd     ?                ;Двойное слово
```

Пример 3

```
;Сегмент команд                     ST           ST(1)    zz
                                ;1.34e-189...      ?       ?
        fst    zz               ;1.34e-189...      ?  1.34e-189
;Сегмент данных
zz      dq     ?                ;Четверное слово
```

FSTCW Запись слова управления сопроцессора

Команда fstcw и аналогичная ей команда fnstcw используют один операнд. В результате выполнения в этот операнд записывается текущее слово управления сопроцессора. При выполнении команды fstcw перед записью состояния производится про-

верка условия немаскируемой ошибки с плавающей точкой. При выполнении команды fnstcw такая проверка не производится. Команда не влияет на флаги.

Пример

```
;Сегмент данных
cword   dw    ?
;Сегмент команд
        fstcw  cword
        fnstcw cword
```

FSTENV Запись рабочей среды сопроцессора

Команда fstenv и аналогичная ей команда fnstenv используют один операнд. В результате выполнения текущая рабочая среда сопроцессора записывается в 14- или 28-байтовый буфер памяти, заданный операндом. Размер операнда определяется тем, какой атрибут use установлен для текущего сегмента кодов. Если установлен атрибут use16, то используется 14-байтовый буфер, в случае use32 – 28-байтоый. Рабочая среда сопроцессора включает в себя слово управления, слово состояния, слово признаков и указатели исключительных ситуаций. При выполнении команды fstenv перед записью состояния производится проверка условия немаскируемой ошибки с плавающей точкой. При выполнении команды fnstenv такая проверка не производится. Команды не влияют на флаги.

Пример

```
;Сегмент данных
buf     db    14 dup (?)
;Сегмент команд
code    segment use16
        fnstenv           buf
```

FSTP Запись действительного числа и выталкивание из стека

При использовании этой команды необходимо указывать один операнд; команда копирует в него содержимое вершины стека. Операндом может быть регистр стека, а также переменная, занимающая двойное или четверное слово. Перед записью выполняется округление в соответствии со значением поля RC управляющего слова до размеров операнда. Если RC=00b, то выполняется округление до ближайшего числа. Если RC=01b или 10b, то число округляется в сторону уменьшения или увеличения соответственно. При RC=11b лишние цифры дробной части числа отбрасываются. После записи производится выталкивание из стека:

```
        fstp  mem        ;mem=ST, выталкивание
        fstp  ST(i)      ;ST(i)=ST, выталкивание
```

Команда влияет на флаги PE, UE, OE, IE.

Пример 1

```
                         ;   ST              ST(1)      ST(2)
                         ;1.2459e1976         12.1        ?
        fstp  ST(2)      ;   12.1          1.2459e1976    ?
```

Пример 2

```
;Сегмент команд              ST           ST(1)    ST(2)    zo
                         ;1.34e-189...    1.9       ?        ?
        fstp  zo         ;  1.9            ?        ?        0
;Сегмент данных
zo      dd    ?          ;Двойное слово
```

Пример 3

```
;Сегмент команд                  ST       ST(1)  ST(2)   zz
                               ;1.34e-189  1.9     ?       ?
        fstp   zo              ;  1.9       ?       ?  1.34e-189
;Сегмент данных
zz      dq     ?               ;Четверное слово
```

FSTSW Запись слова состояния сопроцессора

Команда fstsw и схожая с ней команда fnstsw используют один операнд. В операнд, в качестве которого может быть использовано не только поле данных, определенное как dw, но и регистр AX, записывается текущее слово состояния сопроцессора. При выполнении команды fstsw перед записью слова состояния производится проверка условия немаскируемой ошибки с плавающей точкой. При выполнении команды fnstsw такая проверка не производится. Команды не влияют на флаги.

Пример

```
;Сегмент данных
sword   dw     ?
;Сегмент команд
        fstsw  sword
        fnstsw AX
```

FSUB Вычитание двух действительных чисел

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда вычитает второй операнд из содержимого первого и засылает результат в первый операнд. Операндами могут быть только регистры стека:

```
        fsub   ST,ST(i)        ;ST=ST-ST(i)
        fsub   ST(i),ST        ;ST(i)=ST(i)-ST
```

Если указан один операнд, то он вычитается из содержимого вершины стека, результат помещается в вершину стека. Операндом может быть только переменная, описанная как двойное или четверное слово:

```
        fsub   mem             ;ST=ST-mem
```

При отсутствии операндов вычитается содержимое вершины стека ST из находящейся под ней ячейки ST(1). Результат записывается в ST(1) и после этого выполняется выталкивание из стека:

```
        fsub                   ;ST(1)=ST(1)-ST, выталкивание
```

Команда влияет на флаги PE, UE, OE, DE, IE.

Пример 1

```
;Сегмент команд                 ST     ST(1)  ST(2)  ST(3)
        fld    x1              ; 4.2     ?      ?      ?
        fld    x2              ; 6.3    4.2     ?      ?
        fld    x3              ;10.0    6.3    4.2     ?
        fsub   ST,ST(2)        ; 5.8    6.3    4.2     ?
;Сегмент данных
x1      dd     4.2
x2      dd     6.3
x3      dd     10.0
```

Пример 2

```
;Сегмент команд                 ST     ST(1)  ST(2)  ST(3)
        fld    x1              ; 4.2     ?      ?      ?
```

```
        fld    x2           ; 6.3   4.2     ?      ?
        fld    x3           ;10.0   6.3    4.2     ?
        fsub   ST(2),ST     ;10.0   6.3   -5.8     ?
;Сегмент данных
x1      dd     4.2
x2      dd     6.3
x3      dd     10.0
```

Пример 3

```
;Сегмент команд              ST    ST(1)  ST(2)
        fld    x1           ;11.5    ?      ?
        fld    x2           ; 6.3   11.5    ?
        fsub   ST,ST        ; 0.0   11.5    ?
;Сегмент данных
x1      dd     11.5
x2      dd     6.3
```

Пример 4

```
;Сегмент команд              ST      ST(1)
        fld    um           ;4.0e307  ?
        fsub   wch          ;3.8e307  ?
;Сегмент данных
um      dq     0.4e308      ;Четверное слово
wch     dq     2.0e306      ;Четверное слово
```

Пример 5

```
;Сегмент команд              ST    ST(1)  ST(2)  ST(3)
        fld    x1           ; 6.0    ?      ?      ?
        fld    x2           ; 2.0   6.0     ?      ?
        fld    x3           ; 4.0   2.0    6.0     ?
        fsub                ;-2.0   6.0     ?      ?
;Сегмент данных
x1      dd     6.0
x2      dd     2.0
x3      dd     4.0
```

FSUBP Вычитание двух действительных чисел и запись результата в любой регистр стека с последующим выталкиванием из стека

Существует три варианта использования данной команды: без операндов, с одним операндом и с двумя операндами.

Если указано два операнда, команда вычитает второй операнд из первого и засылает результат в первый операнд. Затем производится выталкивание из стека. Операндами могут быть только регистры стека. В качестве первого операнда запрещается использовать регистр ST:

```
        fsubp  ST(i),ST     ;ST(i)=ST(i)-ST, выталкивание
```

Если указан один операнд, он вычитается из содержимого вершины стека, результат помещается в операнд. Затем производится выталкивание из стека. Операндом может быть только регистр стека. Допускается использовать ST:

```
        fsubp  ST(i)        ;ST(i)=ST(i)-ST, выталкивание
```

При отсутствии операндов вычитается содержимое вершины стека ST из находящейся под ней ячейки ST(1). Результат записывается в регистр ST(1). Затем производится выталкивание из стека:

```
        fsubp               ;ST(1)=ST(1)-ST, выталкивание
```

Команда влияет на флаги PE, UE, OE, DE, IE.

Пример 1

```
;Сегмент команд                ST    ST(1)  ST(2)  ST(3)
        fld    x1           ; 4.0    ?      ?      ?
        fld    x2           ; 1.5   4.0     ?      ?
        fld    x3           ;10.0   1.5    4.0     ?
        fsubp  ST(2),ST     ; 1.5  -6.0     ?      ?
;Сегмент данных
x1      dd     4.0
x2      dd     1.5
x3      dd     10.0
```

Пример 2

```
;Сегмент команд                ST    ST(1)  ST(2)
        fld    f            ; 4.0    ?      ?
        fld    s            ;10.0   4.0     ?
        fsubp  ST(1)        ;-6.0           ?
;Сегмент данных
f       dd     4.0
s       dd     10.0
```

Пример 3

```
;Сегмент команд               ST    ST(1)  ST(2)  ST(3)
        fld    x1           ;6.0     ?      ?      ?
        fld    x2           ;2.0    6.0     ?      ?
        fld    x3           ;1.5    2.0    6.0     ?
        fsubp               ;0.5    6.0     ?      ?
;Сегмент данных
x1      dd     6.0
x2      dd     2.0
x3      dd     1.5
```

FSUBR Вычитание двух действительных чисел в обратном порядке

Данная команда схожа с командой fsub, но отличается от нее тем, что вычитаемое и уменьшаемое меняются местами:

```
        fsubr  ST,ST(i)     ;ST=ST(i)-ST
        fsubr  ST(i),ST     ;ST(i)=ST-ST(i)
        fsubr  mem          ;ST=mem-ST
        fsubr               ;ST(1)=ST-ST(1), выталкивание
```

Команда влияет на флаги PE, UE, OE, DE, IE.

Пример 1

```
;Сегмент команд                ST    ST(1)  ST(2)  ST(3)
        fld    x1           ; 4.2    ?      ?      ?
        fld    x2           ; 6.3   4.2     ?      ?
        fld    x3           ;10.0   6.3    4.2     ?
        fsubr  ST,ST(2)     ;-5.8   6.3    4.2     ?
;Сегмент данных
x1      dd     4.2
x2      dd     6.3
x3      dd     10.0
```

Пример 2

```
;Сегмент команд                ST    ST(1)  ST(2)  ST(3)
        fld    x1           ; 4.2    ?      ?      ?
        fld    x2           ; 6.3   4.2     ?      ?
        fld    x3           ;10.0   6.3    4.2     ?
        fsubr  ST(2),ST     ;10.0   6.3    5.8     ?
;Сегмент данных
x1      dd     4.0
x2      dd     6.3
```

```
x3      dd    10.0
```

Пример 3

```
;Сегмент команд              ST         ST(1)
        fld    wch          ;2.6e-35      ?
        fsubr  wch+8        ;4.94e-34     ?
;Сегмент данных
wch     dq     2.6e-35, 5.2e-34
```

Пример 4

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld    x1           ;6.0      ?       ?       ?
        fld    x2           ;2.0     6.0      ?       ?
        fld    x3           ;4.0     2.0     6.0      ?
        fsubr               ;2.0     6.0      ?       ?
;Сегмент данных
x1      dd     6.0
x2      dd     2.0
x3      dd     4.0
```

FSUBRP Вычитание двух действительных чисел в обратном порядке с последующим выталкиванием из стека

Данная команда схожа с командой fsubp, но отличается от нее тем, что вычитаемое и уменьшаемое меняются местами:

```
        fsubrp ST,ST(i)     ;ST=ST(i)-ST, выталкивание
        fsubrp ST(i)        ;ST(i)=ST(i)-ST, выталкивание
        fsubrp              ;ST(1)=ST-ST(1), выталкивание
```

Команда влияет на флаги PE, UE, OE, DE, IE.

Пример 1

```
;Сегмент команд              ST     ST(1)   ST(2)   ST(3)
        fld    x1           ;6.0      ?       ?       ?
        fld    x2           ;2.0     6.0      ?       ?
        fld    x3           ;4.5     2.0     6.0      ?
        fsubrp ST(2)        ;2.0    -1.5      ?       ?
;Сегмент данных
x1      dd     6.0
x2      dd     2.0
x3      dd     4.5
```

Пример 2

```
;Сегмент команд              ST      ST(1)   ST(2)   ST(3)
        fld    x1           ;6.0       ?       ?       ?
        fld    x2           ;3.7      6.0      ?       ?
        fld    x3           ;4.2      3.7     6.0      ?
        fsubrp              ;0.5      6.0      ?       ?
;Сегмент данных
x1      dd     6.0
x2      dd     3.7
x3      dd     4.2
```

FTST Сравнение вершины стека с нулем

Команда не требует операндов, выполняя сравнение числа из вершины стека с нулем. По результатам сравнения устанавливаются разряды C3, C2 и C0 слова состояния (табл. П-3.3). Команда влияет на флаги DE, IE.

Таблица П-3.3. Значения разрядов C0, C2 и C3 слова состояния сопроцессора после выполнения команды ftst.

Результат сравнения	*C3*	*C2*	*C0*
ST>0.0	0	0	0
ST<0.0	0	0	1
ST=0.0	1	0	0
ST=?	1	1	1

FUCOM Неупорядоченное сравнение вещественных чисел (80387+)

Существует два варианта использования команды: с одним операндом и без операндов. В том и в другом случае эта команда производит неупорядоченное сравнение содержимого вершины стека с содержимым одного из регистров стека. Сравниваемый регистр может быть явно указан в виде операнда:

```
fucom ST(i)        ;Неупорядоченное сравнение ST с ST(i)
```

Если операнд не указан, то содержимое вершины стека сравнивается с содержимым регистра ST(1):

```
fucom              ;Неупорядоченное сравнение ST с ST(1)
```

По результатам сравнения устанавливаются разряды C3, C2 и C0 слова состояния (табл. П-3.4).

Таблица П-3.4. Значения разрядов C0, C2 и C3 слова состояния сопроцессора после выполнения команд неупорядоченного сравнения

Результат сравнения	*C3*	*C2*	*C0*
ST>ST(i)	0	0	0
ST<ST(i)	0	0	1
ST=ST(i)	1	0	0
ST несравнимо с ST(i)	1	1	1

В отличие от команды fucom, команда неупорядоченного сравнения не вызывает исключения "недействительная операция" в случае NAN-операнда. NAN-операнд появляется при выполнении операций занесения данных в занятый регистр стека или чтения из свободного регистра.

Команда влияет на флаги DE, IE.

Пример

```
                   ;ST      ST(1)     ST(2)
                   ;45.7    34.8      127.9
fucom ST(2)        ;C3=0, C2=0, C0=1
fucom              ;C3=0, C2=0, C0=0
```

FUCOMP Неупорядоченное сравнение вещественных чисел с выполнением операции выталкивания числа, находящегося в вершине стека (80387+)

Существует два варианта использования команды: с одним операндом и без операндов. В том и в другом случае эта команда производит неупорядоченное сравнение содержимого вершины стека с содержимым одного из регистров стека. Затем выполняется операция выталкивания из стека. Сравниваемый регистр может быть явно указан в виде операнда

```
fucomp ST(i)       ;Неупорядоченное сравнение ST с ST(i), выталкивание
```

Если операнд не указан, то содержимое вершины стека сравнивается с содержимым ST(1), после чего выполняется операция выталкивания из стека

```
        fucomp          ;Неупорядоченное сравнение ST с ST(1), выталкивание
```

По результатам сравнения устанавливаются разряды C3, C2 и C0 слова состояния (см. табл. П-3.4).

В отличие от команды fucom, команда неупорядоченного сравнения не вызывает исключения "недействительная операция" в случае NAN-операнда. NAN-операнд появляется при выполнении операций занесения данных в занятый регистр стека или чтения из свободного регистра.

Команда влияет на флаги DE, IE.

Пример

```
                        ;ST     ST(1)    ST(2)
                        ;45.3   42.7     42.7
        fucomp ST(2)    ;42.7   42.7      ? C3=0, C2=0, C0=0
        fucomp          ;42.7    ?        ? C3=1, C2=0, C0=0
```

FUCOMPP Неупорядоченное сравнение вещественных чисел и выталкивание сравниваемых чисел из стека (80387+)

Команда fucompp выполняет неупорядоченное сравнение содержимого вершины стека с содержимым регистра ST(1). По результатам сравнения устанавливаются разряды C3, C2 и C0 слова состояния (см. табл. П-3.4). После выполнения операции сравниваемые числа выталкиваются из стека:

```
        fucompp         ;Неупорядоченное сравнение ST с ST(1),
                        ;выталкивание, выталкивание
```

В отличие от команды fucom, команда неупорядоченного сравнения не вызывает исключения "недействительная операция" в случае NAN-операнда. NAN-операнд появляется при выполнении операций занесения данных в занятый регистр стека или чтения из свободного регистра.

Команда влияет на флаги DE, IE.

Пример

```
                        ;ST     ST(1)    ST(2)
                        ;45.3   12.7     42.3
        fucompp         ;42.3    ?        ?  C3=0, C2=0, C0=0
```

FWAIT Ожидание окончания работы сопроцессора

Команда не требует операндов. Она синхронизирует работу основного процессора и сопроцессора, приостанавливая действия процессора до завершения сопроцессором текущей операции. В современных процессорах команды сопроцессора автоматически синхронизируются, т. е. процессор ожидает окончания выполнения предыдущей команды сопроцессора перед запуском следующей. При использовании сопроцессора 8087 для гарантии синхронизации необходима команда fwait. Команда нс влияет на флаги.

FXAM Проверка содержимого вершины стека

Команда не требует операндов. Она возвращает информацию о содержимом вершины стека, устанавливая разряды C3, C2, C1 и C0 слова состояния сопроцессора, в соответствии с таблицей П-3.5. Команда не изменяет флаги.

Таблица П-3.5. Значения разрядов C0, C1, C2 и C3 после выполнения команды fxam

Содержимое вершины стека	*C3*	*C2*	*C1*	*C0*
Положительное ненормализованное число	0	0	0	0
Не число (знак +)	0	0	0	1
Отрицательное ненормализованное число	0	0	1	0
Не число (знак –)	0	0	1	1
Положительное нормализованное число	0	1	0	0
+ бесконечность	0	1	0	1
Отрицательное нормализованное число	0	1	1	0
– бесконечность	0	1	1	1
+ 0.0	1	0	0	0
Пусто	1	0	0	1
– 0.0	1	0	1	0
Пусто	1	0	1	1
Положительное денормализованное число	1	1	0	0
Пусто	1	1	0	1
Отрицательное денормализованное число	1	1	1	0
Пусто	1	1	1	1

FXCH Обмен содержимым двух регистров сопроцессора

Существует два варианта использования данной команды: без операндов и с одним операндом.

Если задан один операнд, то производится обмен содержимым этого операнда и вершины стека. В качестве операнда можно использовать только регистр стека:

```
        fxch   ST(i)         ;Обмен содержимым между ST и ST(i)
```

При отсутствии операндов выполняется обмен содержимым регистров ST и ST(1):

```
        fxch                 ;Обмен содержимым между ST и ST(1)
```

Команда влияет на флаг IE.

Пример

```
                             ;ST           ST(1)         ST(2)
                             ;8.1e4931     12.7        1.0e-1465
        fxch   ST(2)         ;1.0e-1465    12.7        8.1e4931
        fxch                 ;12.7       1.0e-1465     8.1e4931
```

FXTRACT Выделение показателя степени и мантиссы

Команда не требует операндов. Она разделяет число, расположенное в вершине стека, на мантиссу и показатель степени. Вместо числа записывается показатель степени. После этого в стек записывается мантисса. Команда влияет на флаг IE.

Пример 1

```
;Сегмент данных
nmb      dw     8.0          ;Два в третьей степени
;Сегмент команд               ST      ST(1)     ST(2)
        fild   nmb           ;8.0       ?         ?
        fxtract              ;1        3.0        ?
```

Пример 2

```
;Сегмент данных
nmb      dd     14.54
;Сегмент команд               ST         ST(1)     ST(2)
        fild   nmb           ;14.54       ?         ?
        fxtract              ;1.817...   3.0        ?
```

FYL2X Вычисление Y*Log2X

Команда не требует операндов. Она вычисляет произведение числа, записанного в ST(1), на логарифм по основанию 2 числа, записанного в ST. После этого выполняется выталкивание из стека и результат записывается на место сомножителя. Таким образом, после выполнения операции исходные числа теряются. ST не может быть отрицательным. Команда влияет на флаги PE, UE, OE, DE.

Пример 1

```
;Сегмент данных
nmb      dw    8              ;Два в третьей степени
;Сегмент команд               ST      ST(1)     ST(2)
         fld1                 ;1.0      ?         ?
         fild   nmb           ;8.0     1.0        ?
         fyl2x                ;3.0      ?         ?
```

Пример 2

```
;Сегмент данных
nmb      dw    1024           ;Два в десятой степени
;Сегмент команд               ST          ST(1)       ST(2)
         fldpi                ;3.14...      ?           ?
         fild   nmb           ;1024       3.14...       ?
         fyl2x                ;31.4...      ?           ?
```

Пример 3

```
;Сегмент данных
h1       dt    2.0e-128
a12      dd    1269.56
;Сегмент команд               ST                ST(1)       ST(2)
         fld    h1            ;2.0e-128           ?           ?
         fld    a12           ;1269.56         2.0e-128       ?
         fyl2x                ;2.06...e-127       ?           ?
```

Пример 4

```
;Сегмент данных
nmb      dd    1.24
;Сегмент команд               ST               ST(1)    ST(2)
         fld1                 ;1.0               ?        ?
         fld    nmb           ;0.24             1.0       ?
         fyl2xp1              ;0.3103401317      ?        ?
```

FYL2XP1 Вычисление Y*Log2(X+1)

Команда не требует операндов. Она вычисляет произведение числа, записанного в регистр ST(1), на логарифм по основанию 2 от числа, записанного в ST, сложенного с единицей. После этого выполняется выталкивание из стека и результат записывается на место сомножителя. Таким образом после выполнения операции исходные числа теряются. Операнд ST должен находиться в диапазоне sqrt(2)/2–1<= ST<=sqrt(2)–1. Команда влияет на флаги PE, UE, OE, DE, IE.

Пример

```
;Сегмент данных
nmb      dd    0.24
;Сегмент команд               ST               ST(1)    ST(2)
         fld1                 ;1.0               ?        ?
         fld    nmb           ;0.24             1.0       ?
         fyl2xp1              ;0.3103401143      ?        ?
```

Справочные данные по функциям DOS

INT 1Ch. Прерывание, служащее для перехвата прикладной программой тактов системного таймера

Команда INT 1Ch включена в системный обработчик прерываний 8h от системных часов. Системный обработчик прерываний 1Ch фактически выполняет лишь команду IRET; прикладная программа может установить собственный обработчик прерываний 1Ch, который будет активизироваться тактами системного таймера с частотой 18,2 Гц.

INT 20h Завершение программы

Завершает программу типа .COM, возвращая управление родительскому процессу (обычно интерпретатору команд COMMAND). В процессе завершения освобождает всю выделенную процессу память, сбрасывает на диск буферы, закрывает все открытые дескрипторы. Рекомендуется вместо этого прерывания использовать функцию 4Ch прерывания 21h.

При вызове:

CS=сегментный адрес префикса программы PSP

INT 21h, функция 00h. Завершение программы

Завершает программу типа .COM, возвращая управление родительскому процессу (обычно интерпретатору команд COMMAND). В процессе завершения освобождает всю выделенную процессу память, сбрасывает на диск буферы, закрывает все открытые дескрипторы. Рекомендуется вместо этого прерывания использовать функцию 4Ch прерывания 21h.

При вызове:

CS=сегментный адрес префикса программы PSP

INT 21h, функция 01h. Ввод символа с эхом

Вводит символ с клавиатуры и отображает его на экране. При отсутствии символа ждет ввода. Допустимо перенаправление ввода на другое устройство. Если ввод не перенаправлен, выполняет обработку Ctrl+C. Если ввод перенаправлен, выполняет обработку Ctrl+C лишь при включенном режиме BREAK (BREAK=ON). Для чтения расширенного кода ASCII требуется повторное выполнение функции.

При вызове:

AH=01h

При возврате:

AL=код введенного символа

INT 21h, функция 02h. Вывод символа

Выводит символ на экран. Допустимо перенаправление вывода на другое устройство. Выполняет обработку Ctrl+C при вводе этой команды с клавиатуры перед выводом каждого 64-го символа. Коды ASCII: 07h – звонок, 08h – шаг назад, 09 – табуляция, 0Dh – возврат каретки, 0Ah – перевод строки – рассматриваются как управляющие, и выполняются соответствующие им действия.

При вызове:

AH=02h

DL=код выводимого символа

INT 21h, функция 03h. Ввод символа из последовательного порта

Вводит символ из последовательного порта. При отсутствии символа ждет ввода. Допустимо перенаправление ввода на другое устройство.

При вызове:

AH=03h

При возврате:

AL=код выводимого символа

INT 21h, функция 04h. Вывод символа в последовательный порт

Выводит символ в последовательный порт. Если порт, функция ждет его освобождения. Выполняет обработку Ctrl+C при вводе этой команды с клавиатуры.

При вызове:

AH=04h

DL=выводимый байт

INT 21h, функция 05h. Вывод символа на принтер

Выводит символ на принтер. Если принтер занят, функция ждет его освобождения. Выполняет обработку Ctrl+C при вводе этой команды с клавиатуры.

При вызове:

AH=05h

DL=код выводимого символа

INT 21h, функция 06h. Прямой ввод-вывод

Вводит с клавиатуры или выводит символ на экран. В режиме вывода коды ASCII: 07h – звонок, 0Dh – возврат каретки, 0Ah – перевод строки – рассматриваются как управляющие и выполняются соответствующие им действия. Код 08h – возврат на шаг отрабатывается, только если вывод не перенаправлен. Допустимо перенаправление ввода-вывода. Для чтения расширенного кода ASCII требуется повторное выполнение функции. При отсутствии символа не ждет его ввода, а возвращает управление в программу.

При вызове:

AH=06h

DL=код выводимого символа (00h...FEh) (при выводе)

DL=FFh (при вводе)

При возврате:

AL=код введенного символа (при вводе); если символа нет, то ZF=1

INT 21h, функция 07h. Нефильтрованный ввод без эха

Вводит символ с клавиатуры без его отображения на экране. При отсутствии символа ждет ввода. Допустимо перенаправление ввода на другое устройство. Не выполняет обработку Ctrl+C. Для чтения расширенного кода ASCII требуется повторное выполнение функции.

При вызове:

AH=07h

При возврате:

AL=код введенного символа

INT 21h, функция 08h. Ввод символа без эха

Вводит символ с клавиатуры без его отображения на экране. При отсутствии символа ждет ввода. Допустимо перенаправление ввода на другое устройство. Для чтения расширенного кода ASCII требуется повторное выполнение функции. Если ввод не перенаправлен, выполняет обработку Ctrl+C. Если ввод перенаправлен, выполняет обработку Ctrl+C при включенном режиме BREAK.

При вызове:

AH=08h

При возврате:

AL=код введенного символа

INT 21h, функция 09h. Вывод строки

Выводит строку символов на экран. Строка должна заканчиваться символом $. Допустимо перенаправленис вывода на другое устройство. Допустимо использование Esc-последовательностей. Коды ASCII: 07h – звонок, 08h – шаг назад, 0Dh – возврат каретки, 0Ah – перевод строки – рассматриваются как управляющие, и выполняются соответствующие им действия. Выполняет обработку Ctrl+C при вводе этой команды с клавиатуры перед выводом каждого 64-го символа.

При вызове:

AH=09h

DS:DX=адрес выводимой строки

INT 21h, функция 0Ah. Буферизованный ввод с клавиатуры

Вводит строку символов с клавиатуры в буфер пользователя с отображением ее на экране. Строка должна заканчиваться символом возврата каретки (0Dh). Допустимо перенаправление ввода на другое устройство. Если ввод не перенаправлен, выполняет обработку Ctrl+C. Если ввод перенаправлен, выполняет обработку Ctrl+C при включенном режиме BREAK.

При вызове:

AH=0Ah

DS:DX=адрес буфера

При возврате:

Даннные помещены в буфер. Формат буфера:

байт 0 – ожидаемая длина строки

байт 1 – фактическая длина введенной строки

байт 2 и далее – строка, заканчивающаяся 0Dh

INT 21h, функция 0Bh. Проверка состояния ввода

Проверяет наличие символа в буфере клавиатуры. Допустимо перенаправление ввода на другое устройство. Если ввод не перенаправлен, выполняет обработку Ctrl+C. Если ввод перенаправлен, выполняет обработку Ctrl+C при включенном режиме BREAK.

При вызове:

AH=0Bh

При возврате:

AL=00h если символа нет

AL=FFh если символ ждет

INT 21h, функция 0Ch. Очистка входного буфера и ввод

Очищает кольцевой буфер клавиатуры и активизирует указанную в регистре AL функцию ввода. Допустимо перенаправленис ввода на другое устройство.

При вызове:

AH=0Ch

AL=номер требуемой функции ввода. Допустимы функции 01, 07, 08, 0Ah

DS:DX=адрес буфера (если AL=0Ah)

При возврате:

AL=байт входных данных (если при вызове AL=0Ah, данные помещаются в буфер)

INT 21h, функция 0Dh. Сброс диска

Выполняет немедленную запись всех дисковых буферов на диск. Не обновляет информации в каталоге диска.

При вызове:

AH=0Dh

INT 21h, функция 0Eh. Выбор диска

Назначает текущий диск и возвращает число логических дисководов в системе.

При вызове:

AH=0Eh

AL=код дисковода (0=A, 1=B и т. д.)

При возврате:

AL=число логических дисководов в системе

INT 21h, функция 19h. Получение текущего диска

Возвращает код текущего диска.

При вызове:

AH=19h

При возврате:

AL=код текущего диска (0=A, 1=B и т. д.)

INT 21h, функция 1Ah. Установка адреса области обмена с диском

Позволяет определить адрес дисковой области передачи (DTA) для последующих операций с блоками управления файлами.

При вызове:

AH=1Ah

DS:DX=адрес DTA

INT 21h, функция 1Bh. Получение информации о текущем диске

Возвращает характеристики текущсго диска.

При вызове:

AH=1Bh

При возврате:

AL=количество секторов в кластере

CX=количество байтов в секторе

DX=общее количество кластеров на диске

DS:BX=адрес байта описания носителя. Значения этого байта:

FDh -- дискета 360 Кбайт

F9h – дискета 1,2 Мбайт

F8h - жесткий диск

F0h – другие

INT 21h, функция 1Ch. Получение информации о заданном диске

Возвращает характеристики заданного диска.

При вызове:

AH=1Ch

При возврате:

AL=количество секторов в кластере

CX=количество байтов в секторе

DX=общее количество кластеров на диске

DS:BX=адрес байта описания носителя (см. функцию 1Bh)

INT 21h, функция 25h. Установка вектора прерывания

Позволяет заполнить вектор прерывания адресом обработчика прерываний.

При вызове:

AH=25h

AL=номер вектора прерывания

DS:DX=адрес обработчика прерываний

INT 21h, функция 2Ah. Получение текущей даты

Позволяет получить значение текущей даты системного календаря.

При вызове:

AH=2Ah

При возврате:

CX=год (от 1980)

DH=месяц (от 1 до 12)

DL=день (от 1 до 31)

AL=день недели (0 – воскресенье, 1 – понедельник и т. д.)

INT 21h, функция 2Bh. Установка текущей даты

Позволяет изменить текущую дату системного календаря.

При вызове:

AH=2Bh

CX=год (от 1980)

DH=месяц (от 1 до 12)

DL=день (от 1 до 31)

При возврате:

AL=0 – успешное выполнение

AL=FFh – недопустимая дата, системная дата не изменилась)

INT 21h, функция 2Ch. Получение текущего времени

Позволяет получить значение текущего времени системных часов.

При вызове:

AH=2Ch

При возврате:

CH=часы (от 0 до 23)

CL=минуты (от 0 до 59)

DH=секунды (от 0 до 59)

INT 21h, функция 2Dh. Установка текущего времени

Позволяет изменить текущее время системных часов.

При вызове:
AH=2Dh
CH=часы (от 0 до 23)
CL=минуты (от 0 до 59)
DH=секунды (от 0 до 59)
При возврате:
AL=00h – успешное выполнение
AL=FFh – недопустимое время, системное время не изменилось)

INT 21h, функция 2Eh. Установка флага проверки

Изменяет состояние флага проверки записи на диск.

При вызове:
AH=2Eh
AL=00h установить флаг проверки
AL=01h сбросить флаг проверки

INT 21h, функция 2Fh. Получение адреса области обмена с диском

Возвращает адрес текущей дисковой области передачи (DTA).

При вызове:
AH=2Fh
При возврате:
ES:DX=адрес DTA

INT 21h, функция 30h. Получение версии DOS

Возвращает номер используемой версии MS-DOS.

При вызове:
AH=30h
При возврате:
AL=номер версии (например, 6)
AH=номер подверсии (например, 22)

INT 21h, функция 31h. Завершение программы и сохранение ее резидентной в памяти

Завершает выполнение активной программы, резервируя при этом для завершаемой программы указанный объем памяти и возвращая управление родительскому процессу. Возвращает в DOS код возврата. В процессе завершения сбрасывает на диск буферы, закрывает дескрипторы, восстанавливает из ячеек PSP векторы 22h, 23h, 24h.

При вызове:
AH=31h
AL=код возврата
DX=объем резервируемой памяти в параграфах

INT 21h, функция 32h. Получение блока параметров (BPB) для заданного дисковода

Позволяет получить список параметров диска для заданного дисковода.

При вызове:
AH=32h
DL=номер дисковода (00h – по умолчанию, 01h – A: и т. д.)
При возврате:
DS:BX=адрес блока параметров дисковода для заданного дисковода

Формат блока параметров дисковода приведен в табл. П-4.1.

Таблица П-4.1. Формат блока параметров дисковода

Смещение	*Размер, байт*	*Описание*
00h	1	Номер дисковода (00h – A:, 01h – B: и т. д.)
01h	1	Номер устройства в драйвере дисковода
02h	2	Размер сектора в байтах
04h	1	Наибольший номер сектора в кластере
05h	1	Счетчик сдвига для преобразования кластеров в секторы
06h	2	Число резервных секторов в начале дисковода
08h	1	Число таблиц размещения файлов (FAT)
09h	2	Число элементов в корневом каталоге
0Bh	2	Номер первого сектора, содержащего данные пользователя
0Dh	2	Наибольший номер кластера
0Fh	2	Размер FAT в секторах
11h	2	Номер сектора первого каталога
13h	4	Адрес заголовка драйвера устройства
17h	1	Байт идентификации носителя
18h	1	00h, если к диску было обращение; FFh – в противном случае
19h	4	Указатель на следующий DPB в системе
1Dh	2	Кластер, с которого начинается поиск свободного места при записи на диск
1Fh	2	Число свободных кластеров на диске; FFFFh – неизвестно

INT 21h, функция 33h, подфункции 00h и 01h. Получение или установка состояния Break

Позволяет определить или задать условия реакции DOS на ввод с клавиатуры команд Ctrl+C или Ctrl+Break. Функция не использует внутренние стеки DOS и поэтому реентерабельна.

При вызове:

AH=33h

AL=00h получить состояние Break

AL=01h установить состояние Break:

DL=00h состояние Break выключено (OFF)

DL=01h состояние Break включено (ON)

При возврате:

DL=текущее состояние Break (если при вызове AL=00h):

00h состояние Break выключено (OFF)

01h состояние Break включено (ON)

INT 21h, функция 33h, подфункция 02h. Получение и установка состояния Break

Позволяет определить и задать условия реакции DOS на ввод с клавиатуры команды Ctrl+C или Ctrl+Break в одной операции. Функция не использует внутренние стеки DOS и поэтому реентерабельна.

При вызове:

AX=3302h

DL=00h состояние Break выключено (OFF)

DL=01h состояние Break включено (ON)

При возврате:
DL=прошлое состояние Break:
00h состояние Break выключено (OFF)
01h состояние Break включено (ON)

INT 21h, функция 33h, подфункция 05h. Получение дисковода загрузки

Определяет дисковод, с которого была загружена система.

При вызове:
AX=3305h

При возврате:
DL=дисковод загрузки (1=A:, 2=B и т. д.)

INT 21h, функция 34h. Получение адреса флага занятости DOS (флага InDOS)

Возвращает адрес байта области текущих данных DOS (SDA), содержащего флаг занятости DOS (флаг InDOS).

При вызове:
AH=34h

При возврате:
ES:BX=адрес 1-байтового флага InDOS

INT 21h, функция 35h. Получение вектора прерывания

Возвращает содержимое указанного вектора прерывания.

При вызове:
AH=35h
AL=номер вектора прерывания

При возврате:
ES:BX=адрес обработчика прерываний

INT 21h, функция 36h. Получение объема свободного пространства на диске

Возвращает информацию о дисководе, из которой можно вычислить емкость носителя и объем незанятого пространства. Потерянные кластеры считаются занятыми.

При вызове:
AH=36h
DL=код дисковода (00h=текущий, 01h=A: и т. д.)

При возврате:
AX=число секторов в кластере (при ошибке AX=FFFFh)
BX=число свободных кластеров
CX=размер сектора в байтах
DX=полное число кластеров на диске

INT 21h, функция 39h. Создание каталога

Создает каталог в конце указанного пути.

При вызове:
AH=39h
DS:DX=адрес пути в виде строки ASCIIZ

INT 21h, функция 3Ah. Удаление каталога

Удаляет указанный каталог.

При вызове:
AH=3Ah
DS:DX=адрес каталога в виде строки ASCIIZ

INT 21h, функция 3Bh. Смена текущего каталога

Устанавливает новый текущий каталог.

При вызове:

AH=3Bh

DS:DX=адрес каталога в виде строки ASCIIZ

INT 21h, функция 3Ch. Создание или усекновение файла

Создает новый файл с указанной спецификацией. Если указанный файл существует, он усекается до нулевой длины. В любом случае файл открывается и возвращается его дескриптор для дальнейших операций над файлом.

При вызове:

AH=3Ch

CX=атрибуты файла (могут комбинироваться):

1 – только для чтения

2 – скрытый

4 – системный

8 – метка тома

20h – атрибут архива

DS:DX=адрес спецификации файла в виде строки ASCIIZ

При возврате:

AX=дескриптор

INT 21h, функция 3Dh. Открытие файла

Открывает файл с указанной спецификацией. Возвращает дескриптор для последующих операций над файлом. Устанавливает указатель на начало файла (байт 0).

При вызове:

AH=3Dh

AL=режим доступа:

0 – чтение

1 – запись

2 – запись и чтение

Примечание. Если к режиму добавлено 80h, дескриптор наследуется дочерним процессом. В противном случае дескриптор не наследуется дочерним процессом.

DS:DX=адрес спецификации файла в виде строки ASCIIZ

При возврате:

AX=дескриптор

INT 21h, функция 3Eh. Закрытие файла

Сбрасывает на диск внутренние буферы файла, закрывает файл и освобождает дескриптор. Если файл был модифицирован, в записи каталога устанавливаются новые значения длины файла, а также даты и времени создания файла.

При вызове:

AH=3Eh

BX=дескриптор

INT 21h, функция 3Fh. Чтение из файла или устройства

Пересылает из файла или устройства данные в буфер пользователя и модифицирует указатель в файле. При чтении из символьного устройства в режиме ASCII читается строка указанной длины, либо до символа возврата каретки, если он встретился раньше.

При вызове:
AH=3Fh
BX=дескриптор
CX=число пересылаемых байтов
DS:DX=адрес буфера пользователя
При возврате:
AX=число переданных байтов

INT 21h, функция 40h. Запись в файл или в устройство

Пересылает в файл или на устройство данные из буфера пользователя и модифицирует указатель в файле. Если при вызове CX=0, длина файла устанавливается в соответствии с текущим положением указателя. Если перед выводом с клавиатуры вводится Ctrl+C и режим BREAK включен (BREAK=ON), выполняется обработка Ctrl+C.

При вызове:
AH=40h
BX=дескриптор
CX=число пересылаемых байтов
DS:DX=адрес буфера пользователя
При возврате:
AX=число переданных байтов

INT 21h, функция 41h. Удаление файла

Удаляет указанный файл.
При вызове:
AH=41h
DS:DX=спецификация файла в виде строки ASCIIZ

INT 21h, функция 42h. Установка указателя в файле

Позволяет установить текущее положение указателя на любой байт файла для выполнения последующих операций прямого доступа к файлу (чтения или записи).

При вызове:
AH=42h
AL=режим установки указателя:
00h – абсолютное смещение от начала файла
01h – знаковое смещение от текущего положения указателя
02h – знаковое смещение от конца файла
BX=дескриптор
CX=старшая часть смещения
DX=младшая часть смещения
При возврате:
DX=старшая часть возвращенного указателя
AX=младшая часть возвращенного указателя

INT 21h, функция 43h. Получение или установка атрибутов файла

Позволяет получить или изменить значения атрибутов файла или каталога. Файл нельзя преобразовать в каталог или метку тома. Каталог можно сделать скрытым.

При вызове:
AH=43h
AL=00h для получения атрибутов

AL=01h для установки атрибутов
CX=атрибуты файла (могут комбинироваться):
0001h – только для чтения
0002h – скрытый
0004h – системный
0020h – атрибут архивации
DS:DX=адрес спецификации файла или каталога
При возврате:
CX=возвращаемые атрибуты файла (если при вызове AL=0)

INT 21h, функция 45h. Дублирование дескриптора файла

Создает новый дескриптор файла, который связан с заданным файлом или устройством через тот же элемент системной таблицы файлов (SFT).

При вызове:
AH=45h
BX=дескриптор файла
При возврате:
AX=новый дескриптор

INT 21h, функция 46h. Принудительное дублирование дескриптора файла

Принудительно объявляет указанный дескриптор дубликатом заданного. Если дескриптор в BX был открыт, он закрывается.

При вызове:
AH=46h
BX=дескриптор файла
CX=дескриптор, который должен стать дубликатом первого

INT 21h, функция 47h. Получение текущего каталога

Возвращает строку ASCIIZ с полным путем (от корневого каталога) к текущему каталогу, включая его имя. Не возвращается обозначение текущего дисковода и корневого каталога (знак \).

При вызове:
AH=47h
DL=код дисковода (0=текущий, 1=A: и т. д.)
DS:SI=адрес буфера размером 64 байта
При возврате:
Буфер заполнен спецификацией текущего каталога

INT 21h, функция 48h. Выделение блока памяти

Выделяет блок памяти и возвращает его сегментный адрес.

При вызове:
AH=48h
BX=требуемое число параграфов памяти
При возврате:
AX=сегментный адрес выделенного блока

Если память не выделена, CF=1, AX=код ошибки, BX=размер наибольшего доступного блока памяти в параграфах

INT 21h, функция 49h. Освобождение блока памяти

Освобождает блок памяти и передает его системе.

При вызове:
AH=49h
ES=сегментный адрес освобождаемого блока

INT 21h, функция 4Ah. Изменение размера выделенного блока памяти

Уменьшает или увеличивает размер выделенного блока памяти.

При вызове:
AH=4Ah
BX=требуемый размер блока в параграфах
ES=сегментный адрес модифицируемого блока

Если память не выделена, CF=1, AX=код ошибки, BX=размер наибольшего доступного блока памяти в параграфах

INT 21h, функция 4Bh. Запуск программы (функция Exec)

Позволяет родительскому процессу, в частности активной прикладной программе, загрузить и запустить другую программу (дочерний процесс). После завершения запущенной программы управление возвращается родительскому процессу.

При вызове:
AH=4Bh
AL=00h загрузить и выполнить программу
AL=01h загрузить и не выполнять программу
AL=03h загрузить оверлей
ES:BX=адрес блока параметров. Формат блока параметров:

```
dw      envirseg          ;Сегмент окружения
dd      cmdtail           ;Адрес хвоста команды
dd      0,0               ;Адреса FCB(в настоящее время не используются)
```

DS:DX=адрес спецификации запускаемой программы в виде строки ASCIIZ

INT 21h, функция 4Ch. Завершение процесса с кодом возврата

Завершает текущий процесс (программу), помещая указанный код завершения в предназначенный для него байт области текущих данных DOS (SDA). В процессе завершения освобождает всю выделенную процессу память, сбрасывает на диск буферы, закрывает все открытые дескрипторы, из ячеек PSP восстанавливает векторы 22h, 23h и 24h.

При вызове:
AH=4Ch
AL=код возврата

INT 21h, функция 4Dh. Получение кода возврата и типа завершения

Используется родительским процессом после возврата из дочернего процесса, активизированного функцией Exec (INT 21h, функция 4Bh), для получения из области текущих данных DOS (SDA) кодов возврата и завершения процесса. Код возврата в SDA в результате использования данной функции очищается, поэтому ее можно вызывать только однажды.

При вызове:
AH=4Dh

При возврате:
AH=тип завершения
00h – нормальное завершение с помощью INT 20h или INT 21h (функции 00h или 4Ch)
01h – пользователь ввел Ctrl+C
02h – завершение через драйвер критической ошибки

03h – завершение с помощью INT 21h (функции 31h или 27h)
AL=код возврата, передаваемый из дочернего процесса.

INT 21h, функция 4Eh. Нахождение первого файла, соответствующего заданной спецификации

Осуществляет поиск в указанном каталоге первого файла, соответствующего указанному шаблону групповой операции. Если CX=0, ищутся только нормальные файлы (без атрибутов). Если CX=8, ищется только метка тома. При указании других атрибутов или их комбинаций ищутся файлы с указанными атрибутами и нормальные файлы.

При вызове:
AH=4Eh
CX=атрибуты искомых файлов (могут комбинироваться)
1 – только для чтения
2 – скрытый
4 – системный
8 – метка тома
10h – каталог
20h – атрибут архивации
DS:DX=адрес спецификации искомого файла
При возврате:
Имя файла и расширение заносятся в область обмена с диском в байты 1Eh...2Ah

INT 21h, функция 4Fh. Нахождение следующего файла

Осуществляет поиск следующего файла после того, как функция 4Eh нашла первый файл, соответствующий указанному шаблону групповой операции. Используется только после успешного выполнения функции 4Eh. Если предполагается, что файлов, соответствующих шаблону, может быть больше двух, функцию следует выполнять многократно до получения CF=1 (файлов, соответствующих шаблону, больше нет).

При вызове:
AH=4Fh
При возврате:
Имя файла и расширение заносятся в область обмена с диском в байты 1Eh...2Ah; если следующий файл не найден, CF=1

INT 21h, функция 50h. Установка идентификатора текущего процесса

Позволяет записать в область текущих данных DOS адрес PSP программы, которую требуется объявить текущей. Функция не использует внутренние стеки DOS и поэтому реентерабельна.

При вызове:
AH=50h
BX=сегментный адрес PSP процесса, объявляемого текущим

INT 21h, функция 51h. Получение идентификатора текущего процесса

Позволяет получить адрес PSP программы, которую DOS считает текущей. Функция не использует внутренние стеки DOS и поэтому реентерабельна.

При вызове:
AH=51h
При возврате:
BX=сегментный адрес PSP текущего процесса

INT 21h, функция 52h. Получение адреса списка списков

Возвращает адрес "списка списков" – базовой системной таблицы, содержащей информацию о ряде других таблиц DOS.

При вызове:

AH=52h

При возврате:

ES:BX=адрес списка списков

INT 21h, функция 54h. Получение флага проверки

Позволяет получить состояние флага DOS проверки правильности записи на диск.

При вызове:

AH=54h

При возврате:

AL=флаг проверки:

00h=выключен

01h=включен

INT 21h, функция 56h. Переименование файла

Переименовывает файл или перемещает его в другой каталог на том же диске. Допустимо переименование каталога. Недопустимо использование шаблонов групповых операций.

При вызове:

AH=56h

DS:DX=адрес текущей спецификации файла

ES:DI=адрес новой спецификации файла

INT 21h, функция 5700h. Получение даты и времени создания или модификации файла

Позволяет получить дату и время создания файла, записанные в каталоге. Файл должен быть предварительно создан или открыт.

При вызове:

AX=5700h

BX=дескриптор файла

При возврате:

CX=новое время. Биты 0...4 – 2-секундные интервалы, 5h...Ah – минуты, Bh...Fh – часы

DX=дата. Биты 0...4 – день, 5...8 – месяц, 9h...Fh – год относительно 1980

INT 21h, функция 5701h. Установка даты и времени создания файла

Позволяет модифицировать дату и время создания файла, записанные в каталоге. Файл должен быть предварительно создан или открыт.

При вызове:

AX=5701h

BX=дескриптор файла

CX=новое время (см. функцию 5700h)

DX=новая дата (см. функцию 5700h)

INT 21h, функция 5Ah. Создание временного файла

Создает файл с указанными атрибутами в указанном каталоге и возвращает дескриптор и имя файла. Имя файлу назначается системой. При завершении программы

файл не удаляется. Функцию удобно использовать, если в программе требуется создать большое и неопределенное заранее количество файлов, конкретные имена которых не имеют особого значения.

При вызове:

AH=5Ah

CX=атрибуты файла (могут комбинироваться):

1 – только для чтения

2 – скрытый

4 – системный

20h – атрибут архива

DS:DX=адрес спецификации каталога в виде строки ASCIIZ

При возврате:

AX=дескриптор файла

DS:DX=адрес полной спецификации файла в виде строки ASCIIZ

INT 21h, функция 5Bh. Создание нового файла

Создает новый файл с указанной спецификацией и атрибутами и возвращает дескриптор. Если указанный файл уже существует, функция завершается с ошибкой.

При вызове:

AH=5Bh

CX=атрибуты файла (могут комбинироваться; см. функцию 5Ah)

DS:DX=адрес спецификации файла в виде строки ASCIIZ

При возврате:

AX=дескриптор файла

INT 21h, функция 5D06h. Получение адреса области текущих данных DOS

Возвращает адрес области текущих данных DOS (Swappable Data Area, SDA), в которой хранится ряд системных переменных, и в частности находятся все три стека DOS.

При вызове:

AX=5D06h

При возврате:

DS:SI=адрес SDA

CX=размер в байтах части SDA, которая должна сохраняться при переходе на другой процесс, если прерывается функция DOS

DX=размер в байтах части SDA, которая должна сохраняться при переходе на другой процесс во всех случаях

INT 21h, функция 62h. Получение идентификатора текущего процесса

Позволяет получить адрес PSP программы, которую DOS считает текущей. Функция не использует внутренние стеки DOS и поэтому реентерабельна. Идентична функции 51h

При вызове:

AH=62h

При возврате:

BX=сегментный адрес PSP текущего процесса

INT 21h, функция 67h. Установка числа дескрипторов

Устанавливает максимальное число файлов и устройств, которые могут быть одновременно открыты текущим процессом. Фактически функция создает новую табли-

цу файлов задания JFT, копируя в ее начало исходную JFT, находящуюся в PSP программы и имеющую размер 20 байт. Новая таблица создается в свободной памяти за пределами программы, и для ее успешного выполнения требуется, чтобы в системе был свободный блок памяти соответствующего размера. Поскольку максимальное число открытых файлов лимитируется не только размером JFT, но так же и числом блоков описания файлов в системной таблице файлов SFT, наряду с расширением JFT требуется также расширить SFT с помощью директивы файла CONFIG.SYS FILES=.

При вызове:

AH=67h

BX=требуемое число дескрипторов

При возврате:

В PSP текущей программы записан адрес новой JFT

INT 21h, функция 68h. Сброс буферов DOS в файл

Выполняет принудительное обновление файла на диске. Все данные, находящиеся в буферах DOS, записываются в файл. Обновляется запись каталога.

При вызове:

AH=68h

BX=дескриптор

INT 21h, функция 6900h. Получение серийного номера тома

Читает метку тома и серийный номер тома для данного диска.

При вызове:

AX=6900hh

BL=дисковод (0=текущий, 1=A: и т. д.)

DS:BX=адрес буфера размером 26 байт

При возврате:

В буфер помещается копия содержимого расширенного блока параметров BIOS (BPB) на диске

Формат буфера приведен в табл. П-4.2.

Таблица П-4.2. Формат расширенного блока параметров диска

Смещение	*Число байтов*	*Содержимое*
00h	2	0
02h	4	Серийный номер диска
06h	11	Метка тома или 'NONAME ', если она отсутствует
11h	8	Тип файловой системы – строки 'FAT12' или 'FAT16'

INT 21h, функция 6901h. Установка серийного номера тома

Записывает метку тома и серийный номер тома для данного диска.

При вызове:

AX=6901h

BL=дисковод (0=текущий, 1=A: и т. д.)

DS:BX=адрес буфера размером 18 байт

При возврате:

В расширенный BPB на диске копируется информация из байтов 0...10h буфера (см. функцию 6900h)

INT 23h. Обработчик Ctrl+C – Ctrl+Break

Служит для обработки команд пользователя на аварийное завершение текущей программы. При вводе пользователем команды Ctrl+C в кольцевой буфер клавиатуры поступает код 2E03h; в случае ввода команды Ctrl+Break кольцевой буфер очищается, а в буферный байт драйвера клавиатуры CON засылается код 03h. Большинство функций DOS перед выполнением закрепленного за ними действия анализируют наличие кода 03h как в кольцевом буфере клавиатуры, так и в буфере драйвера. При обнаружении этого кода функция DOS выполняет команду INT 23h. Системный обработчик этого прерывания завершает текущую программу вызовом функции DOS 4Ch. Прикладная программа, установив собственный обработчик прерывания 23h, может вмешаться в процесс аварийного завершения программы, включив в него действия, необходимые для аккуратного завершения программы.

INT 25h. Абсолютное чтение с диска

Позволяет прочитать в память с диска один или группу секторов с заданным начальным относительным номером. Секторы нумеруются с нуля от начала логического (не физического!) диска. Таким образом, загрузочный сектор данного логического диска имеет номер 0, следующий за ним на диске первый сектор первой копии FAT – номер 1 и т. д.

После выполнения прерываний int 25h и int 26h в стеке задачи остается слово, содержащее значение регистра флагов. Если это слово не удалить, то нарушится дальнейший ход программы.

При вызове:

AL=номер дисковода (0=A, 1=B и т. д.)
CX=число читаемых секторов
DX=относительный номер первого читаемого сектора
DS:BX=адрес буфера

При ошибке:

CF=1, AX=коды ошибки:

Код ошибки в регистре AH:

01h – неправильная команда
02h – неправильная адресная метка
04h – запрошенный сектор не найден
08h – ошибка прямого доступа к памяти
10h – ошибка данных (неправильная контрольная сумма)
20h – ошибка контроллера
40h – ошибка позиционирования

Код ошибки в регистре AL:

00h – ошибка защиты записи
01h – неизвестное устройство
02h – дисковод не готов
03h – неизвестная команда
04h – ошибка данных (неправильная контрольная сумма)
06h – ошибка позиционирования
07h – неизвестный тип носителя
08h – сектор не найден

INT 26h. Абсолютная запись на диск

Позволяет записать из памяти на диск один или группу секторов с заданным начальным относительным номером. Секторы нумеруются с нуля от начала логического (не физического!) диска. Таким образом, загрузочный сектор данного логического диска имеет номер 0, следующий за ним на диске первый сектор первой копии FAT – номер 1 и т. д.

После выполнения прерываний int 25h и int 26h в стеке задачи остается слово, содержащее значение регистра флагов. Если это слово не удалить, то нарушится дальнейший ход программы.

При вызове:

AL=номер дисковода (0=A, 1=B и т. д.)

CX=число читаемых секторов

DX=относительный номер первого читаемого сектора

DS:BX=адрес буфера

При ошибке:

CF=1, AX=коды ошибки (см. примечание к INT 25h)

INT 2Fh. Мультиплексное прерывание

Прерывание предназначено для организации связи между процессами, и в частности для обмена информацией с системными и прикладными резидентными программами. Для пользователя зарезервированы функции C0h...FFh.

При вызове:

AH=функция

AL=подфункция

Другие регистры используются по мере необходимости

При возврате:

AL=0 если программа не установлена и можно приступить к ее установке

AL=FFh если программа уже установлена

Коды ошибок при выполнении функций DOS

Если DOS не смогла выполнить требуемую функцию, перед возвратом в вызывающую программу DOS устанавливает флаг CF в регистре флагов, а в регистр AX заносит код ошибки. В таблице П-4.3 приведены наиболее употребительные коды ошибок.

Таблица П-4.3. Коды ошибок DOS

Код	*Описание*	*Код*	*Описание*
01h	Неправильный номер функции	15h	Дисковод не готов
02h	Файл не найден	16h	Неизвестная команда
03h	Путь не найден	17h	Ошибка контрольной суммы
04h	Слишком много открытых файлов	18h	Неверная длина структуры запроса
05h	Доступ запрещен	19h	Ошибка поиска дорожки
06h	Неправильный дескриптор	1Ah	Неизвестный носитель
07h	Уничтожен блок управления памятью	1Bh	Сектор не найден
08h	Нехватка памяти	1Ch	В принтере нет бумаги
09h	Неправильный адрес блока памяти	1Dh	Отказ записи
0Ah	Неправильное окружение	1Eh	Отказ чтения
0Bh	Неправильный формат	1Fh	Общая ошибка
0Ch	Неправильный код доступа	20h	Нарушение разделения
0Dh	Неправильные данные	21h	Нарушение запирания файла

0Eh	Неизвестное устройство	22h	Недопустимая смена дискеты
0Fh	Неправильный дисковод	50h	Файл уже существует
10h	Попытка удалить текущий каталог	52h	Каталог не может быть создан
11h	Не то же устройство	53h	Отказ по прерыванию Int 21h (критическая ошибка)
12h	Больше нет файлов	54h	Слишком много перенаправлений
13h	Диск с защитой от записи	55h	Двойное перенаправление
14h	Неизвестное устройство	57h	Неправильный параметр

Справочные данные по прерываниям BIOS

Видеосистема, прерывание 10h

INT 10h, функция 00h. Установка видеорежима

Устанавливает текущий видеорежим.

При вызове:

AH=00h

AL=видеорежим:

03h текстовый, 80x25, 16 цветов

10h графический, 640x350, 16 цветов (EGA)

12h графический, 640x480, 16 цветов (VGA)

1Dh графический, 800x600, 256 цветов (VGA)

38h графический, 1024x768, 256 цветов (VGA)

INT 10h, функция 01h. Установка конфигурации курсора

Позволяет задать начальную и конечную строки развертки мерцающего аппаратного курсора в текстовых видеорежимах.

При вызове:

AH=01h

CH биты 0...4 = начальная строка развертки курсора

CL биты 0...4 = конечная строка развертки курсора

INT 10h, функция 02h. Установка позиции курсора

Задает положение курсора на экране в текстовых координатах на указанной странице (в том числе неактивной). Курсор можно установить как в текстовом, так и в графическом режиме, однако в графическом режиме курсор не виден.

При вызове:

AH=02h

BH=страница

DH=строка

DL=столбец

INT 10h, функция 03h. Получение позиции и размера курсора

Возвращает положение курсора на экране для заданной страницы (в том числе неактивной).

При вызове:

AH=03h

BH=страница

При возврате:

CH=начальная строка развертки для курсора

CL=конечная строка развертки для курсора

DH=строка

DL=столбец

INT 10h, функция 05h. Установка видеостраницы

Устанавливает активную видеостраницу (как текстовую, так и графическую).

При вызове:
AH=05h
AL=страница

INT 10h, функция 06h. Инициализация или прокрутка окнавверх

Инициализирует окно с указанными координатами пробелами ASCII с заданным атрибутом или прокручивает содержимое окна вверх на заданное число строк. Действует только для активной страницы. При прокрутке появляющиеся внизу строки заполняются пробелами ASCII с заданным атрибутом. Функцию удобно использовать для быстрой очистки всего экрана или любой прямоугольной области на экране.

При вызове:
AH=06h
AL=число строк прокрутки; если AL=0, все окно очищается
BH=атрибут символов в окне
CH=y-координата верхнего левого угла окна
CL=x-координата верхнего левого угла окна
DH=y-координата нижнего правого угла окна
DL=x-координата нижнего правого угла окна

INT 10h, функция 07h. Инициализация или прокрутка окна вниз

Инициализирует окно с указанными координатами пробелами ASCII с заданным атрибутом или прокручивает содержимое окна вниз на заданное число строк. Действует только для активной страницы. При прокрутке появляющиеся вверху строки заполняются пробелами ASCII с заданным атрибутом. Функцию удобно использовать для быстрой очистки всего экрана или любой прямоугольной области на экране.

При вызове:
AH=07h
AL=число строк прокрутки; если AL=0, все окно очищается
BH=атрибут символов в окне
CH=y-координата верхнего левого угла окна
CL=x-координата верхнего левого угла окна
DH=y-координата нижнего правого угла окна
DL=x-координата нижнего правого угла окна

INT 10h, функция 08h. Чтение символа и атрибута в позиции курсора

Возвращает символ ASCII и его атрибут в позиции курсора на заданной странице (не только активной).

При вызове:
AH=08h
BH=страница
При возврате:
AH=атрибут
AL=символ

INT 10h, функция 09h. Запись символа и атрибута в позицию курсора

Записывает символ и его атрибут в текущую позицию курсора как в графическом, так и в текстовом режиме. В графическом режиме символы не должны переходить на следующую строку. Все коды в AL рассматриваются как знаки и не управляют положением курсора. После вывода символа курсор нужно сместить к следующей позиции

функцией 02h. Коэффициент повторения позволяет выводить строки одинаковых символов (но курсор не смещается). В текстовом режиме символ выводится с указанным атрибутом, т. е. заданного цвета на заданном фоне. В графическом режиме содержимое BL влияет на цвет только символа, но не фона под ним. Однако графическое изображение под знакоместом затирается.

При вызове:
AH=09h
AL=символ
BH=страница
BL=атрибут (текстовый режим) или цвет (графический режим)
CX=коэффициент повторения

INT 10h, функция 0Ah. Запись символа в позицию курсора

Записывает символ ASCII в текущую позицию курсора как в графическом, так и в текстовом режиме. Символ принимает атрибут, установленный ранее для этой позиции. В графическом режиме символы не должны переходить на следующую строку. Все коды в AL рассматриваются как знаки и не управляют положением курсора. После вывода символа курсор следует сместить к следующей позиции функцией 02h. Коэффициент повторения позволяет выводить строки одинаковых символов (но курсор не смещается).

При вызове:
AH=0Ah
AL=символ
BH=страница
CX=коэффициент повторения

INT 10h, функция 0Ch. Запись пиксела

Записывает в видеобуфер точку заданного цвета в заданной графической позиции.

При вызове:
AH=0Ch
AL=цвет (номер цветового регистра)
BH=страница
CX=графический столбец
DX=графическая строка

INT 10h, функция 0Dh. Чтение пиксела.

Читает из видеобуфера цвет пиксела в заданной графической позиции.

При вызове:
AH=0Ch
BH=страница
CX=графический столбец
DX=графическая строка
При возврате:AL=цвет (номер цветового регистра)

INT 10h, функция 0Eh. Запись символа в режиме телетайпа

Записывает символ ASCII в текущую позицию курсора на активной странице и смещает курсор к следующей позиции. Коды ASCII: 07h – звонок, 08h – шаг назад, 0Dh – возврат каретки, 0Ah – перевод строки – рассматриваются как управляющие, и выполняются соответствующие им действия. Остальные управляющие коды рас-

сматриваются как знаки и выводятся на экран. Действует автоматический перевод курсора на следующую строку и скроллинг экрана. Атрибут символа указать нельзя; при записи действует атрибут, установленный ранее для текущей позиции.

При вызове:

AH=0Eh

AL=символ

BL=цвет символа (в графическом режиме)

INT 10h, функция 0Fh. Получение видеорежима

Позволяет получить текущий видеорежим видеоконтроллера.

При вызове:

AH=0Fh

При возврате:

AH=число символьных столбцов на экране

AL=видеорежим

BH=активная видеостраница

INT 10h, функция 10h, подфункция 00h. Настройка цветового регистра

Устанавливает соответствие номера цветового регистра цвету пиксела. Каждый цветовой регистр содержит 6 значащих разрядов, которые определяют интенсивность красного, зеленого и синего компонентов, дающих при смешивании требуемый цвет. Разряды 0...2 закреплены за цветами красный, зеленый, синий с интенсивностью 2/3 максимальной; разряды 3...5 – за теми же цветами с интенсивностью 2/3. Таким образом, число 1 (С), записанное в регистр, определяет синий цвет, число 9 (с+С) – ярко-синий, число 7 (К+З+С) – белый, число FFh (к+з+с+К+З+С) – ярко-белый. Регистр, содержащий 0, определяет черный цвет.

При вызове:

AX=1000h

BH=значение цвета в коде кзсКЗС (см. описание подфункции 00h)

BL=номер цветового регистра (0...15)

INT 10h, функция 10h, подфункция 01h. Установка цвета края экрана

Устанавливает цвет края экрана (выбегов развертки).

При вызове:

AX=1001h

BH=значение цвета в коде кзсКЗС (см. описание подфункции 00h)

INT 10h, функция 10h, подфункция 02h. Настройка цветовой палитры и установка цвета края экрана

Устанавливает соответствие номеров цветовых регистров цветам пикселов, а также цвет края экрана (выбегов развертки).

При вызове:

AX=1002h

ES:DX=адрес 17-байтовой таблицы цветов.

Таблица заполняется кодами кзсКЗС (см. описание подфункции 00h), загружаемыми в цветовые регистры 0...15 (первые 16 байт) и в регистр края экрана (последний байт).

INT 10h, подфункция 03h. Переключение назначения бита "мерцаниеяркость"

Определяет назначение старшего бита (7) атрибута символа: мерцание символа или повышенная яркость фона под ним.

При вызове:

AX=1003h

BL=назначение старшего бита байта атрибута:

0 – повышенная яркость фона

1 – мерцание символа

INT 10h, функция 10h, подфункция 07h.

Чтение цветового регистра.

При вызове:

AX=1007h

При возврате:

BH=значение цвета в коде кзсКЗС (см. описание подфункции 00h)

INT 10h, функция 10h, подфункция 09h. Чтение цветовой палитры и цвета края экрана

Позволяет одной командой получить содержимое всех 17 цветовых регистров.

При вызове:

AX=1009h

ES:DX=адрес 17-байтового буфера для содержимого цветовых регистров

INT 10h, функция 11h, подфункция 03h. Задание спецификатора блока знакогенератора

Позволяет определить назначение бита 3 байта атрибутов символа в текстовых видеорежимах: яркость символа или номера блоков знакогенератора. Биты 0...1 регистра BL определяют номер блока (от 0 до 3) знакогенератора, символы из которого поступают на экран, если бит 3 байта атрибутов равен нулю. Биты 2...3 регистра BL определяют номер блока (от 0 до 3) знакогенератора, символы из которого поступают на экран, если бит 3 байта атрибутов равен единице. Если значения битов 0... 1 и 2...3 совпадают, используется только один блок знакогенератора (256 символов), а бит 3 байта атрибутов управляет яркостью символов.

При вызове:

AX=1103h

BL=код блока знакогенератора

INT 10h, функция 11h, подфункция 10h. Загрузка шрифта пользователя

Загружает таблицу с определением шрифта пользователя в указанный блок генератора символов. Перепрограммирует контроллер на новый размер шрифта. При использовании подфункции 10h должна быть активна видеостраница 0. Подфункция 00h выполняет те же действия, но не перепрограммирует видеоконтроллер; при использовании этой подфункции активной может быть любая видеостраница.

При вызове:

AX=1110h

BH=высота символа в числе графических точек

BL=блок генератора

CX=число символов, описанных в таблице

DX=код, назначаемый первому символу таблицы

ES:BP=адрес таблицы

INT 10h, функция 11h, подфункция 11h. Загрузка шрифта ПЗУ 8x14

Загружает шрифт ПЗУ BIOS размером 8x14 графических точек, используемый по умолчанию, в указанный блок генератора символов. При использовании подфункции

11h должна быть активна видеостраница 0. Подфункция 01h выполняет те же действия, но не перепрограммирует видеоконтроллер; при использовании этой подфункции активной может быть любая видеостраница.

При вызове:
AH=1111h
BL=блок генератора

INT 10h, функция 11h, подфункция 12h. Загрузка шрифта ПЗУ 8x8

Загружает шрифт ПЗУ BIOS размером 8x8 графических точек, используемый по умолчанию, в указанный блок генератора символов. При использовании подфункции 12h должна быть активна видеостраница 0. Подфункция 02h выполняет те же действия, но не перепрограммирует видеоконтроллер; при использовании этой подфункции активной может быть любая видеостраница.

При вызове:
AX=1112h
BL=блок генератора

INT 10h, функция 11h, подфункция 21h. Установка вектора 43h на шрифт пользователя

Загружает в вектор 43h адрес таблицы шрифтов пользователя для применения в графическом режиме. Одновременно модифицируется область видеоданных BIOS.

При вызове:
AX=1121h
BL=код числа строк на экране
00h=указывается пользователем (см. регистр DL)
01h=14 строк
02h=25 строк
03h=43 строки
CX=число строк пикселов (байтов) на символ
DL=число строк на экране (если BL=00h)
ES:BP=адрес таблицы шрифтов пользователя

INT 10h, функция 11h, подфункция 22h. Установка вектора 43h на шрифт ПЗУ 8x14

Загружает в вектор 43h из ПЗУ BIOS адрес таблицы шрифтов с размером матрицы 8x14 точек для использования в графическом режиме. Одновременно модифицируется область видеоданных BIOS.

При вызове:
AX=1122h

INT 10h, функция 11h, подфункция 23h. Установка вектора 43h на шрифт ПЗУ 8x8

Загружает в вектор 43h из ПЗУ BIOS адрес таблицы шрифтов с размером матрицы 8x8 точек для использования в графическом режиме. Одновременно модифицируется область видеоданных BIOS.

При вызове:
AX=1123h

INT 10h, функция 11h, подфункция 30h. Получение информации о шрифтах

Позволяет получить адреса таблиц шрифтов, а также число строк пикселов (байтов) на символ для данного шрифта.

При вызове:
AX=1130h
BH=код шрифта
00h=текущее содержимое вектора 1Fh
01h=текущее содержимое вектора 43h
02h=шрифт ПЗУ 8x14
03h=шрифт ПЗУ 8x8 (1-я половина)
04h=шрифт ПЗУ 8x8 (2-я половина)
При возврате:
CX=число строк пикселов (байтов) на символ
DL=номер последней строки на экране
ES:BP=адрес таблицы со шрифтом

INT 10h, функция 13h. Запись строки в режиме телетайпа

Записывает строку в текущую страницу видеобуфера начиная с указанной позиции. Коды ASCII: 07h – звонок, 08h – шаг назад, 09h – табуляция, 0Ah – перевод строки, 0Dh – возврат каретки – рассматриваются как управляющие, и выполняются соответствующие им действия.

При вызове:
AH=13h
AL=режим записи:

0 – атрибут в BL, строка содержит только коды символов, курсор не смещается после записи

1 – атрибут в BL, строка содержит только коды символов, курсор смещается после записи

2 – строка содержит попеременно коды символов и атрибутов; курсор не смещается после записи

3 – строка содержит попеременно коды символов и атрибутов; курсор смещается после записи

BH=страница
BL=атрибут (если AL=0 или 1)
CX=длина символьной строки (в длину входят только коды символов, но не байты атрибутов)
DH=номер строки на экране
DL=номер столбца на экране
ES:BP=адрес строки

Дисковая система, прерывание 13h

INT 13h, функция 00h. Сброс дисковой системы

Приводит дисковый контроллер в исходное состояние, позиционирует головки на цилиндр 0 и подготавливает систему в вводу-выводу.

При вызове:
AH=00h
DL=дисковод
00h...7Fh – гибкий диск
80h...FFh – жесткий диск

При ошибке:
CF=1, AH=состояние:
00h – отсутствие ошибки
01h – неправильная команда
02h – не найдена адресная метка
03h – дискета защищена от записи
04h – сектор не найден
05h – сброс жесткого диска не прошел
06h – дискета вынута
07h – неправильная таблица параметров жесткого диска
0Ch – не найден тип носителя данных
0Dh – неправильное число секторов в формате на жестком диске
10h – невосстановимая ошибка данных
11h – восстановленная ошибка данных на жестком диске
20h – неисправность контроллера
40h – ошибка позиционирования
80h – тайм-аут диска
AAh – жесткий диск не готов
BBh – неизвестная ошибка жесткого диска

INT 13h, функция 02h. Чтение секторов

Читает один или группу секторов с физического (не логического!) диска в память. Для начального сектора указываются абсолютные координаты (цилиндр, сектор, головка). Секторы физического диска нумеруются на каждой дорожке с единицы, цилиндры нумеруются с нуля, головки нумеруются с нуля. Сначала идут секторы 1...n цилиндра 0, головки (поверхности) 0, затем секторы 1...n цилиндра 0, головки (поверхности) 1, далее секторы 1...n цилиндра 1, головки 0 и т. д. Таким образом, сектор 1 цилиндра 0 головки 0 относится к главной загрузочной записи (Master boot).

При вызове:
AH=02h
AL=число читаемых секторов
CH=цилиндр
CL=начальный сектор (биты 0...5) и 2 старших бита номера цилиндра (биты 6...7)
DH=головка
DL=дисковод
00h...7Fh – гибкий диск
80h...FFh – жесткий диск
ES:BX=адрес буфера
При возврате:
CF=0, AH=0, AL=число переданных секторов
При ошибке:
CF=1, AH=состояние (см. функцию 00h)

INT 13h, функция 03h. Запись секторов

Записывает один или группу секторов из памяти на физический (не логический!) диск. Для начального сектора указываются абсолютные координаты (цилиндр, сектор, головка). Секторы физического диска нумеруются на каждой дорожке с единицы, цилиндры нумеруются с нуля, головки нумеруются с нуля. Сначала идут секторы 1...n

цилиндра 0, головки (поверхности) 0, затем секторы 1...n цилиндра 0, головки (поверхности) 1, далее секторы 1...n цилиндра 1, головки 0 и т. д. Таким образом, сектор 1 цилиндра 0 головки 0 относится к главной загрузочной записи (Master boot).

При вызове:
AH=03h
AL=число записываемых секторов
CH=цилиндр
CL=начальный сектор (биты 0...5) и 2 старших бита номера цилиндра (биты 6...7)
DH=головка
DL=дисковод
00h...7Fh – гибкий диск
80h...FFh – жесткий диск
ES:BX=адрес буфера
При возврате:
CF=0, AH=0, AL=число переданных секторов
При ошибке:
CF=1, AH=состояние (см. функцию 00h)

INT 13h, функция 05h. Форматирование дорожки на гибком диске

Форматирует указанную дорожку, нанося на нее форматные метки. При форматировании дорожки дискеты следует составить список адресных полей. Каждое поле состоит из 4 байт, в которых указываются: цилиндр, головка, сектор, код размера сектора. Число адресных полей равно числу секторов на дорожке. Код размера сектора может принимать значения:
0 – 128 байт на сектор
1 – 256 байт на сектор
2 – 512 байт на сектор
3 – 1024 байта на сектор.
При вызове:
AH=05h
AL=число форматируемых секторов
CH=цилиндр
DH=головка
DL=дисковод (00h...7Fh)
ES:BX=адрес списка адресных полей
При ошибке:
CF=1, AH=состояние (см. функцию 00h)

Последовательный ввод-вывод, прерывание 14h

INT 14h, функция 00h. Инициализация порта

Задает операционные параметры для конкретного последовательного порта.
При вызове:
AH=00h
AL=параметры порта (см. табл. П-5.1)
DX=номер порта (00h...03h)
При возврате:
AH=состояние линии (см. табл. П-5.2)
AL=состояние модема (см. табл. П-5.2)

Таблица П-5.1. Параметры инициализации последовательного порта

Биты 7–6–5: скорость передачи, бод	*Биты 4...3: контроля четности (паритет)*	*Бит 2: количество стоп-битов*	*Биты 2...10: длина слова, бит*
000: 110	00: отсутствует	0: 1	00: 5
001: 150	01: нечетность	1: 2	01: 6
010: 300	10: отсутствует	–	10: 7
011: 600	11: четность	–	11: 8
100: 1200	–	–	–
101: 2400	–	–	–
110: 4800	–	–	–
111: 9600	–	–	–

Таблица П-5.2. Возвращаемое состояние последовательного порта

Регистр AH		***Регистр AL***	
Биты	*Значение*	*Биты*	*Значение*
7	Тайм-аут	7	Обнаружение сигнала (CD)
6	Передача "Сдвиговый регистр пуст"	6	Индикатор вызова (RI)
5	Индикатор вызова (RI)	5	Модем готов (DSR)
4	Обнаружен Break	4	Готов к передаче (CTS)
3	Ошибка кадра	3	Изменение состояния линии "Обнаружение сигнала"
2	Ошибка четности	2	Спад сигнала вызова
1	Ошибка наложения	1	Изменение состояния линии "Модем готов"
0	Принимаемые данные готовы	0	Изменение состояния линии "Готов к передаче"

INT 14h, функция 01h. Вывод в порт

Выводит один символ в указанный последовательный порт.

При вызове:

AH=01h

AL=выводимый символ

DX=номер порта (00h...03h)

При возврате:

AH бит 7 сброшен, если успешное выполнение

AH бит 7 установлен в случае ошибки

AH биты 6...0 – состояние порта (см. функцию 00h)

INT 14h, функция 02h. Ввод из порта

Вводит один символ из указанного последовательного порта, ожидая в случае необходимости.

При вызове:

AH=02h

DX=номер порта (00h...03h)

При возврате:

AH=состояние линии (см. функцию 00h)

AL=принятый символ, если бит 7 AH сброшен

INT 14h, функция 03h. Получение состояния порта

Читает состояние указанного последовательного порта.

При вызове:

AH=03h

DX=номер порта (00h...03h)

При возврате:

AH=состояние линии (см. функцию 00h)

AL=состояние модема (см. функцию 00h)

Клавиатура, прерывание 16h

INT 16h, функция 00h. Чтение символа с клавиатуры

Читает из кольцевого буфера ввода символ и скан-код. Если буфер пуст, ожидает ввода. Функция отбрасывает расширенные коды ASCII, возвращая значение только в случае поступления обычного, нерасширенного кода.

При вызове:

AH=00h

При возврате:

AH=скан-код

AL=символ ASCII

INT 16h, функция 01h. Получение состояния клавиатуры

Определяет, имеются ли в кольцевом буфере ожидающие ввода символы; возвращает флаг ожидания и сам символ при его наличии. При этом и символ и скан-код не извлекаются из кольцевого буфера и будут снова получены при вызове функции 00h прерывания INT 16h. Однако в процессе проверки наличия ожидающего кода все расширенные коды удаляются из буфера.

При вызове:

AH=01h

При возврате:

Если символ ожидает:

ZF=0, AH=скан-код, AL=код ASCII символа

Если ожидающих символов нет:

ZF=1

INT 16h, функция 02h. Получение флагов клавиатуры

Возвращает байт флагов клавиатуры, описывающий состояние управляющих клавиш клавиатуры (байт в области данных BIOS по адресу 0000h:0417h).

При вызове:

AH=02h

При возврате:

AL=флаги. Биты байта имеют следующие значения:

0 – нажата правая клавиша Shift

1 – нажата левая клавиша Shift

2 – нажата клавиша Ctrl

3 – нажата клавиша Alt

4 – включен режим Scroll Lock

5 – включен режим Num Lock

6 – включен режим Caps Lock

7 – включен режим Insert

INT 16h, функция 03h. Установление задержки и скорости автоповтора

Задает характеристики автоповтора при длительном удерживании любой клавиши в нажатом состоянии.

При вызове:
AH=03h
AL=05h
BH=код значения задержки
BL=код скорости автоповтора
Значения кодов задержки:
00h=1/4 с
01h=1/2 с
02h=3/4 с
03h=1 с
Значения кодов скорости автоповтора:
00h=30 символ/с
01h=26,7 символ/с
02h=24 символ/с
03h=21,8 символ/с
...
1Fh=2 символ/с

INT 16h, функция 05h. Помещение кода символа в буфер клавиатуры

Эмулирует ввод с клавиатуры под управлением программы, засылая указанный символ в кольцевой буфер клавиатуры.

При вызове:
AH=05h
CH=скан-код
CL=код ASCII символа
При возврате:
AL=00h, если успешное выполнение
AL=01h, если буфер клавиатуры полон

INT 16h, функция 10h. Получение кода клавиши расширенной клавиатуры

Читает из кольцевого буфера ввода символ и скан-код. Если буфер пуст, ожидает ввода. В отличие от функции 00h эта функция не отбрасывает расширенные коды ASCII

При вызове:
AH=10h
При возврате:
AH=скан-код
AL=символ ASCII

INT 16h, функция 11h. Получение состояния расширенной клавиатуры

Определяет, имеются ли в кольцевом буфере ожидающие ввода символы; возвращает флаг ожидания и сам символ при его наличии. Однако символ и скан-код не извлекаются из кольцевого буфера и будут снова получены при вызове функции 10h прерывания INT 16h. В отличие от функции 01h эта функция не отбрасывает расширенные коды ASCII.

При вызове:
AH=11h
При возврате:
Если символ ожидает:
ZF=0, AH=скан-код, AL=код ASCII символа
Если ожидающих символов нет:
ZF=1

INT 16h, функция 12h. Получение флагов расширенной клавиатуры

Возвращает всю информацию о флагах клавиатуры, описывающих состояние управляющих клавиш клавиатуры (байты в области данных BIOS по адресам 0000h:0417h и 0000h:0418h).

При вызове:
AH=12h
При возврате:
AL=байт 1 флагов расширенной клавиатуры. Биты байта имеют следующие значения:
0 – нажата правая клавиша Shift
1 – нажата левая клавиша Shift
2 – нажата любая клавиша Ctrl
3 – нажата любая клавиша Alt
4 – включен режим Scroll Lock
5 – включен режим Num Lock
6 – включен режим Caps Lock
7 – включен режим Insert

AH=байт 2 флагов расширенной клавиатуры. Биты байта имеют следующие значения:
0 – нажата левая клавиша Ctrl
1 – нажата левая клавиша Alt
2 – нажата правая клавиша Ctrl
3 – нажата правая клавиша Alt
4 – нажата клавиша Scroll Lock
5 – нажата клавиша Num Lock
6 – нажата клавиша Caps Lock
7 – нажата клавиша SysRq

Принтер, прерывание 17h

INT 17h, функция 00h. Вывод символа

Выводит один символ на принтер.

При вызове:
AH=00h
AL=выводимый символ
DX=номер порта принтера (00h...02h)
При возврате:
AH=состояние принтера:
бит 0: тайм-аут
бит 3: ошибка ввода-вывода
бит 4: выбран

бит 5: нет бумаги
бит 6: подтверждение
бит 7: не занят

INT 17h, функция 01h. Инициализация порта принтера

Сбрасывает порт принтера.

При вызове:

AH=01h

DX=номер порта принтера (00h...02h)

При возврате:

AH=состояние принтера (см. функцию 00h)

INT 14h, функция 02h. Получение состояния порта принтера

Читает состояние указанного принтера.

При вызове:

AH=02h

DX=номер порта принтера (00h...02h)

При возврате:

AH=состояние принтера (см. функцию 00h)

Начальный загрузчик, прерывание 19h

INT 19h. Перезагрузка системы

Перезагружает систему, не очищая память и не восстанавливая векторы прерываний.

Часы реального времени, прерывание 1Ah

INT 1Ah, функция 00h. Получение системного времени

Читает системное время, хранящееся в 4-байтовой ячейкс области данных BIOS по адресу 0040h:006Eh. Содержимое этой ячейки инкрементируется прерыванием от аппаратного таймера через вектор 08h каждые 55 мс (приблизительно 18,2 раза в секунду).

При вызове:

AH=00h

При возврате:

CX:DX=число тактов системного таймера от полуночи

AL=флаг полуночи; не нуль, если с момента последнего чтения время прошло через полночь

INT 1Ah, функция 01h. Установка системного времени

Устанавливает значение системного времени, которое хранится в 4-байтовой ячейке области данных BIOS по адресу 0040h:006Eh. Содержимое этой ячейки инкрементируется прерыванием от аппаратного таймера через вектор 08h каждые 55 мс (приблизительно 18,2 раза в секунду).

При вызове:

AH=01h

CX:DX=число тактов системного таймера от полуночи

INT 1Ah, функция 02h. Получение времени от КМОП-часов реального времени

Читает показания часов реального времени, питающихся от аккумулятора и работающих независимо от состояния включения компьютера.

При вызове:
AH=02h
При возврате:
CF сброшен, если успешное выполнение. В этом случае:
CH=часы в коде BCD
CL=минуты в коде BCD
DH=секунды в коде BCD
DL=флаг летнего времени (00h – стандартное время, 01h – летнее время)
CF установлен, если часы не идут или находятся в процессе обновления времени

INT 1Ah, функция 03h. Установка времени в КМОП-часах реального времени

Устанавливает показания часов реального времени, питающихся от аккумулятора и работающих независимо от состояния включения компьютера.

При вызове:
AH=03h
CH=часы в коде BCD
CL=минуты в коде BCD
DH=секунды в коде BCD
DL=флаг летнего времени (00h – стандартное время, 01h – летнее время)

INT 1Ah, функция 04h. Получение даты от КМОП-календаря реального времени

Читает дату из календаря реального времени, питающегося от аккумулятора и работающего независимо от состояния включения компьютера.

При вызове:
AH=04h
При возврате:
CF сброшен, если успешное выполнение. В этом случае:
CH=две старшие цифры года в коде BCD
CL=две младшие цифры года в коде BCD
DH=месяц в коде BCD
DL=день в коде BCD
CF установлен, если часы не идут или находятся в процессе обновления времени

INT 1Ah, функция 05h. Установка даты в КМОП-календаре реального времени

Читает дату из календаря реального времени, питающегося от аккумулятора и работающего независимо от состояния включения компьютера.

При вызове:
AH=05h
CH=две старшие цифры года в коде BCD
CL=две младшие цифры года в коде BCD
DH=месяц в коде BCD
DL=день в коде BCD

INT 1Ah, функция 06h. Установка будильника в КМОП-часах реального времени

Устанавливает будильник в часах реального времени. В установленный момент КМОП-микросхема возбуждает аппаратное прерывание через вектор 70h, в системном обработчике которого имеется команда INT 4Ah. Прикладная программа, обрабатывающая сигнал от будильника, может перехватывать как аппаратное (70h), так и про-

граммное (4Ah) прерывание. После установки будильник будет возбуждать прерывания каждые сутки до своего выключения функцией 07h.

При вызове:
AH=06h
CH=часы в коде BCD
CL=минуты в коде BCD
DH=секунды в коде BCD

INT 1Ah, функция 07h. Отмена будильника в КМОП-часах реального времени

Запрещает работу будильника в часах реального времени.

При вызове:
AH=07h

INT 4Ah. Прерывание, служащее для перехвата прикладной программой сигнала от будильника в часах реального времени

Команда INT 4Ah включена в системный обработчик прерывания 70h от будильника в часах реального времени. Системный обработчик этого прерывания фактически выполняет лишь команду IRET; прикладная программа может установить собственный обработчик прерывания 4Ah, который будет активизироваться сигналом будильника.

Предметный указатель

Р

Оглавление